Workload Analysis of a Large-Scale Key-Value Store

Berk Atikoglu
Stanford, Facebook
atikoglu@stanford.edu

Yuehai Xu
Wayne State, Facebook
yhxu@wayne.edu

Eitan Frachtenberg
Facebook
etc@fb.com

Song Jiang
Wayne State
sjiang@wayne.edu

Mike Paleczny
Facebook
mpal@fb.com

ABSTRACT

Key-value stores are a vital component in many scale-out enterprises, including social networks, online retail, and risk analysis. Accordingly, they are receiving increased attention from the research community in an effort to improve their performance, scalability, reliability, cost, and power consumption. To be effective, such efforts require a detailed understanding of realistic key-value workloads. And yet little is known about these workloads outside of the companies that operate them. This paper aims to address this gap.

To this end, we have collected detailed traces from Facebook's Memcached deployment, arguably the world's largest. The traces capture over 284 billion requests from five different Memcached use cases over several days. We analyze the workloads from multiple angles, including: request composition, size, and rate; cache efficacy; temporal patterns; and application use cases. We also propose a simple model of the most representative trace to enable the generation of more realistic synthetic workloads by the community.

Our analysis details many characteristics of the caching workload. It also reveals a number of surprises: a GET/SET ratio of 30:1 that is higher than assumed in the literature; some applications of Memcached behave more like persistent storage than a cache; strong locality metrics, such as keys accessed many millions of times a day, do not always suffice for a high hit rate; and there is still room for efficiency and hit rate improvements in Memcached's implementation. Toward the last point, we make several suggestions that address the exposed deficiencies.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Databases; D.4.8 [**Performance**]: Modeling and Prediction; D.4.2 [**Storage Management**]: Distributed Memories

Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMETRICS'12, June 11–15, 2012, London, England, UK.
Copyright 2012 ACM 978-1-4503-1097-0/12/06 ...$10.00.

Keywords

Workload Analysis, Workload modeling, Key-Value Store

1. INTRODUCTION

Key-value (KV) stores play an important role in many large websites. Examples include: Dynamo at Amazon [15]; Redis at GitHub, Digg, and Blizzard Interactive [27]; Memcached at Facebook, Zynga and Twitter [18, 26]; and Voldemort at Linkedin [1]. All these systems store ordered $(key, value)$ pairs and are, in essence, a distributed hash table.

A common use case for these systems is as a layer in the data-retrieval hierarchy: a cache for expensive-to-obtain values, indexed by unique keys. These values can represent any data that is cheaper or faster to cache than re-obtain, such as commonly accessed results of database queries or the results of complex computations that require temporary storage and distribution.

Because of their key role in large website performance, KV stores are carefully tuned for low response times and high hit rates. But like all caching heuristics, a KV-store's performance is highly dependent on its workload. It is therefore imperative to understand the workload's characteristics. Additionally, analyzing and understanding large-scale cache workloads can also: provide insights into topics such as the role and effectiveness of memory-based caching in distributed website infrastructure; expose the underlying patterns of user behavior; and provide difficult-to-obtain data and statistical distributions for future studies.

In this paper, we analyze five workloads from Facebook's Memcached deployment. Aside from the sheer scale of the site and data (over 284 billion requests over a period of 58 sample days), this case study also introduces to the community several different usage scenarios for KV stores. This variability serves to explore the relationship between the cache and various data domains: where overall site patterns are adequately handled by a generalized caching infrastructure, and where specialization would help. In addition, this paper offers the following key contributions and findings:

1. A workload decomposition of the traces that shows how different applications of Memcached can have extreme variations in terms of read/write mix, request sizes and rates, and usage patterns (Sec. 3).
2. An analysis of the caching characteristics of the traces and the factors that determine hit rates. We found that different Memcached pools can vary significantly in their locality metrics, but surprisingly, the best predictor of hit rates is actually the pool's size (Sec. 6).

3. An examination of various performance metrics over time, showing diurnal and weekly patterns (Sec. 3.3, 4.2.2, 6).

4. An analytical model that can be used to generate more realistic synthetic workloads. We found that the salient size characteristics follow power-law distributions, similar to other storage and Web-serving systems (Sec. 5).

5. An exposition of a Memcached deployment that can shed light on real-world, large-scale production usage of KV-stores (Sec. 2.2, 8).

The rest of this paper is organized as follows. We begin by describing the architecture of Memcached, its deployment at Facebook, and how we analyzed its workload. Sec. 3 presents the observed experimental properties of the trace data (from the request point of view), while Sec. 4 describes the observed cache metrics (from the server point of view). Sec. 5 presents a simple analytical model of the most representative workload. The next section brings the data together in a discussion of our results, followed by a section surveying previous efforts on analyzing cache behavior and workload analysis.

2. MEMCACHED DESCRIPTION

2.1 Architecture

Memcached[1] is a simple, open-source software package that exposes data in RAM to clients over the network. As data size grows in the application, more RAM can be added to a server, or more servers can be added to the network. Additional servers generally only communicate with clients. Clients use consistent hashing [9] to select a unique server per *key*, requiring only the knowledge of the total number of servers and their IP addresses. This technique presents the entire aggregate data in the servers as a unified distributed hash table, keeps servers completely independent, and facilitates scaling as data size grows.

Memcached's interface provides the basic primitives that hash tables provide—insertion, deletion, and retrieval—as well as more complex operations built atop them.

Data are stored as individual items, each including a key, a value, and metadata. Item size can vary from a few bytes to over $100\,KB$, heavily skewed toward smaller items (Sec. 3). Consequently, a naïve memory allocation scheme could result in significant memory fragmentation. To address this issue, Memcached adopts a slab allocation technique, in which memory is divided into slabs of different sizes. The slabs in a class store items whose sizes are within the slab's specific range. A newly inserted item obtains its memory space by first searching the slab class corresponding to its size. If this search fails, a new slab of the class is allocated from the heap. Symmetrically, when an item is deleted from the cache, its space is returned to the appropriate slab, rather than the heap. Memory is allocated to slab classes based on the initial workload and its item sizes, until the heap is exhausted. Consequently, if the workload characteristics change significantly after this initial phase, we may find that the slab allocation is inappropriate for the workload, resulting in memory underutilization.

[1] http://memcached.org/

Table 1: Memcached pools sampled (in one cluster). These pools do not match their UNIX namesakes, but are used for illustrative purposes here instead of their internal names.

Pool	Size	Description
USR	few	user-account status information
APP	dozens	object metadata of one application
ETC	hundreds	nonspecific, general-purpose
VAR	dozens	server-side browser information
SYS	few	system data on service location

A new item arriving after the heap is exhausted requires the eviction of an older item in the appropriate slab. Memcached uses the Least-Recently-Used (LRU) algorithm to select the items for eviction. To this end, each slab class has an LRU queue maintaining access history on its items. Although LRU decrees that any accessed item be moved to the top of the queue, this version of Memcached coalesces repeated accesses of the same item within a short period (one minute by default) and only moves this item to the top the first time, to reduce overhead.

2.2 Deployment

Facebook relies on Memcached for fast access to frequently-accessed values. Web servers typically try to read persistent values from Memcached before trying the slower backend databases. In many cases, the caches are demand-filled, meaning that generally, data is added to the cache after a client has requested it and failed.

Modifications to persistent data in the database often propagate as deletions (invalidations) to the Memcached tier. Some cached data, however, is transient and not backed by persistent storage, requiring no invalidations.

Physically, Facebook deploys front-end servers in multiple datacenters, each containing one or more *clusters* of varying sizes. Front-end clusters consist of both Web servers, running primarily HipHop [31], and caching servers, running primarily Memcached. These servers are further subdivided based on the concept of *pools*. A pool is a partition of the entire key space, defined by a prefix of the key, and typically represents a separate application or data domain. The main reason for separate domains (as opposed to one all-encompassing cache) is to ensure adequate quality of service for each domain. For example, one application with high turnover rate could evict keys of another application that shares the same server, even if the latter has high temporal locality but lower access rates. Another reason to separate domains is to facilitate application-specific capacity planning and performance analysis.

In this paper, we describe traces from five separate pools—one trace from each pool (traces from separate machines in the same pool exhibit similar characteristics). These pools represent a varied spectrum of application domains and cache usage characteristics (Table 1). One pool in particular, ETC, represents general cache usage of multiple applications, and is also the largest of the pools; the data collected from this trace may be the most applicable to general-purpose KV-stores.

The focus of this paper is on workload characteristics, patterns, and relationships to social networking, so the exact details of server count and components have little relevance

here. It is important to note, however, that all Memcached instances in this study ran on identical hardware.

2.3 Tracing Methodology

Our analysis called for complete traces of traffic passing through Memcached servers for at least a week. This task is particularly challenging because it requires nonintrusive instrumentation of high-traffic volume production servers. Standard packet sniffers such as tcpdump[2] have too much overhead to run under heavy load. We therefore implemented an efficient packet sniffer called *mcap*. Implemented as a Linux kernel module, mcap has several advantages over standard packet sniffers: it accesses packet data in kernel space directly and avoids additional memory copying; it introduces only 3% performance overhead (as opposed to tcpdump's 30%); and unlike standard sniffers, it handles out-of-order packets correctly by capturing incoming traffic after all TCP processing is done. Consequently, mcap has a complete view of what the Memcached server sees, which eliminates the need for further processing of out-of-order packets. On the other hand, its packet parsing is optimized for Memcached packets, and would require adaptations for other applications.

The captured traces vary in size from $3TB$ to $7TB$ each. This data is too large to store locally on disk, adding another challenge: how to offload this much data (at an average rate of more than $80,000$ samples per second) without interfering with production traffic. We addressed this challenge by combining local disk buffering and dynamic offload throttling to take advantage of low-activity periods in the servers.

Finally, another challenge is this: how to effectively process these large data sets? We used Apache HIVE[3] to analyze Memcached traces. HIVE is part of the Hadoop framework that translates SQL-like queries into MapReduce jobs. We also used the Memcached "stats" command, as well as Facebook's production logs, to verify that the statistics we computed, such as hit rates, are consistent with the aggregated operational metrics collected by these tools.

3. WORKLOAD CHARACTERISTICS

This section describes the observed properties of each trace in terms of the requests that comprise it, their sizes, and their frequencies.

3.1 Request Composition

We begin by looking at the basic data that comprises the workload: the total number of requests in each server, broken down by request types (Fig. 1). Several observations delineate the different usage of each pool:

USR handles significantly more GET requests than any of the other pools. GET operations comprise over 99.8% of this pool's workload. One reason for this is that the pool is sized large enough to maximize hit rates, so refreshing values is rarely necessary. These values are also updated at a slower rate than some of the other pools. The overall effect is that USR is used more like RAM-based persistent storage than a cache.

APP has high GET rates too—owing to the popularity of this application—but also a large number of DELETE operations. DELETE operations occur when a cached database entry is modified (but not required to be set again in the cache). SET operations occur when the Web servers add a value to the cache. The relatively high number of DELETE operations show that this pool represents database-backed values that are affected by frequent user modifications.

ETC has similar characteristics to APP, but with an even higher rate of DELETE requests (of which some may not be currently cached). ETC is the largest and least specific of the pools, so its workloads might be the most representative to emulate. Because it is such a large and heterogenous workload, we pay special attention to this workload throughout the paper.

VAR is the only pool sampled that is write-dominated. It stores short-term values such as browser-window size for opportunistic latency reduction. As such, these values are not backed by a database (hence, no invalidating DELETEs are required). But they change frequently, accounting for the high number of UPDATEs.

SYS is used to locate servers and services, not user data. As such, the number of requests scales with the number of servers, not the number of user requests, which is much larger. This explains why the total number of SYS requests is much smaller than the other pools'.

It is interesting to note that the ratio of GETs to UPDATEs in ETC (approximately $30:1$) is significantly higher than most synthetic workloads typically assume (Sec. 7). For demand-filled caches like USR, where each miss is followed by an UPDATE, the ratios of GET to UPDATE operations mentioned above are related to hit rate in general and the sizing of the cache to the data in particular. So in theory, one could justify any synthetic GET to UPDATE mix by controlling the cache size. But in practice, not all caches or keys are demand-filled, and these caches are already sized to fit a real-world workload in a way that successfully trades off hit rates to cost.

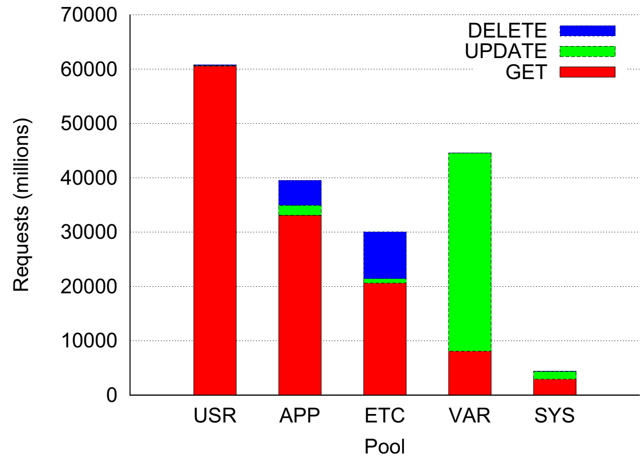


Figure 1: Distribution of request types per pool, over exactly 7 days. UPDATE commands aggregate all non-DELETE writing operations, such as SET, REPLACE, etc.

[2] http://www.tcpdump.org/
[3] http://hive.apache.org/

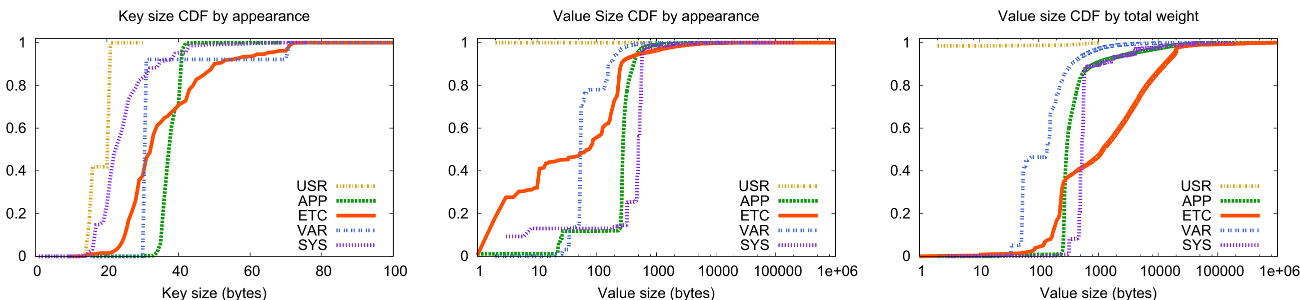


Figure 2: Key and value size distributions for all traces. The leftmost CDF shows the sizes of keys, up to Memcached's limit of $250\,B$ (not shown). The center plot similarly shows how value sizes distribute. The rightmost CDF aggregates value sizes by the total amount of data they use in the cache, so for example, values under $320\,B$ or so in SYS use virtually no space in the cache; $320\,B$ values weigh around 8% of the data, and values close to $500\,B$ take up nearly 80% of the entire cache's allocation for values.

3.2 Request Sizes

Next, we look at the sizes of keys and values in each pool (Fig. 2), based on SET requests. All distributions show strong modalities. For example, over 90% of APP's keys are 31 bytes long, and values sizes around $270\,B$ show up in more than 30% of SET requests. USR is the most extreme: it only has two key size values ($16\,B$ and $21\,B$) and virtually just one value size ($2\,B$). Even in ETC, the most heterogeneous of the pools, requests with 2-, 3-, or 11-byte values add up to 40% of the total requests. On the other hand, it also has a few very large values (around $1MB$) that skew the weight distribution (rightmost plot in Fig. 2), leaving less caching space for smaller values.

Small values dominate all workloads, not just in count, but especially in overall weight. Except for ETC, 90% of all cache space is allocated to values of less than $500\,B$. The implications for caching and system optimizations are significant. For example, network overhead in the processing of multiple small packets can be substantial, which explains why Facebook coalesces as many requests as possible in as few packets as possible [9]. Another example is memory fragmentation. The strong modality of each workload implies that different Memcached pools can optimize memory allocation by modifying the slab size constants to fit each distribution. In practice, this is an unmanageable and unscalable solution, so instead Memcached uses many (44) slab classes with exponentially growing sizes, in the hope of reducing allocation waste, especially for small sizes.

3.3 Temporal Patterns

To understand how production Memcached load varies over time, we look at each trace's transient request rate over its entire collection period (Fig. 3). All traces clearly show the expected diurnal pattern, but with different values and amplitudes. If we increase our zoom factor further (as in the last plot), we notice that traffic in ETC bottoms out around 08:00 and has two peaks around 17:00 and 03:00. Not surprisingly, the hours immediately preceding 08:00 UTC (midnight in Pacific Time) represent night time in the Western Hemisphere.

The first peak, on the other hand, occurs as North America starts its day, while it is evening in Europe, and continues until the later peak time for North America. Although different traces (and sometimes even different days in the same trace) differ in which of the two peaks is higher, the entire period between them, representing the Western Hemisphere day, sees the highest traffic volume. In terms of weekly patterns, we observe a small traffic drop on most Fridays and Saturdays, with traffic picking up again on Sundays and Mondays.

The diurnal cycle represents load variation on the order of $2\times$. We also observe the presence of traffic spikes. Typically, these can represent a swift surge in user interest on one topic, such as occur with major news or media events. Less frequently, these spikes stem from programmatic or operational causes. Either way, the implication for Memcached development and deployment is that one must budget individual node capacity to allow for these spikes, which can easily double or even triple the normal peak request rate. Although such budgeting underutilizes resources during normal traffic, it is nevertheless imperative; otherwise, the many Web servers that would take to this sudden traffic and fail to get a prompt response from Memcached, would all query the same database nodes. This scenario could be debilitating, so it must remain hypothetical.

4. CACHE BEHAVIOR

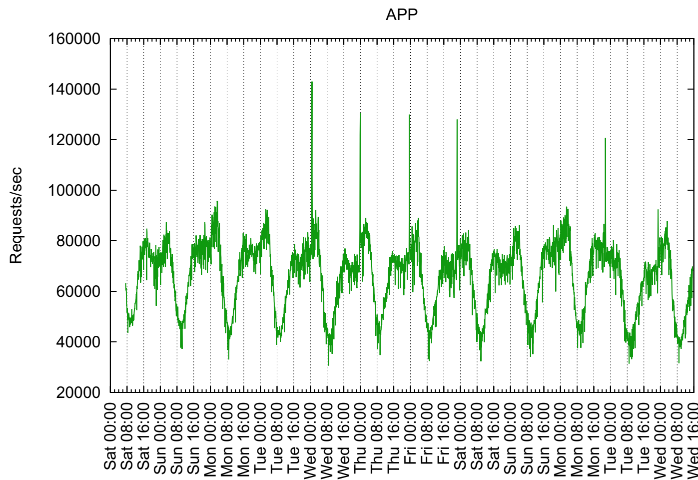
The main metric used in evaluating cache efficacy is hit rate: the percentage of GET requests that return a value. The overall hit rate of each server, as derived from the traces and verified with Memcached's own statistics, are shown in Table 2. This section takes a deeper look at the factors that influence these hit rates and how they relate to cache locality, user behavior, temporal patterns, and Memcached's design.

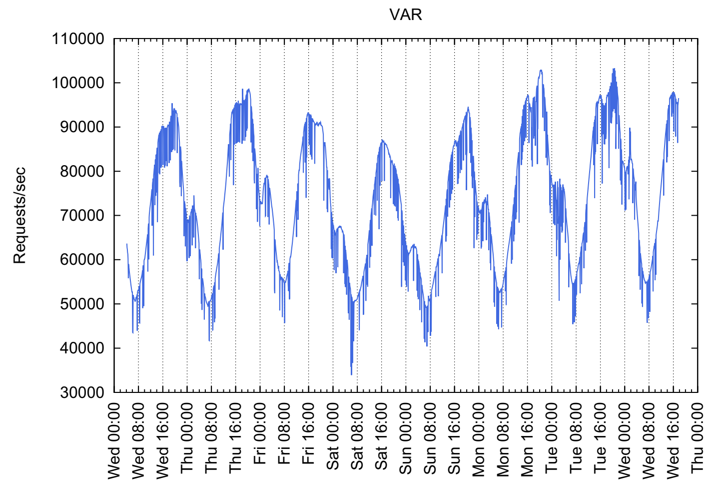
Table 2: Mean cache hit rate over entire trace.

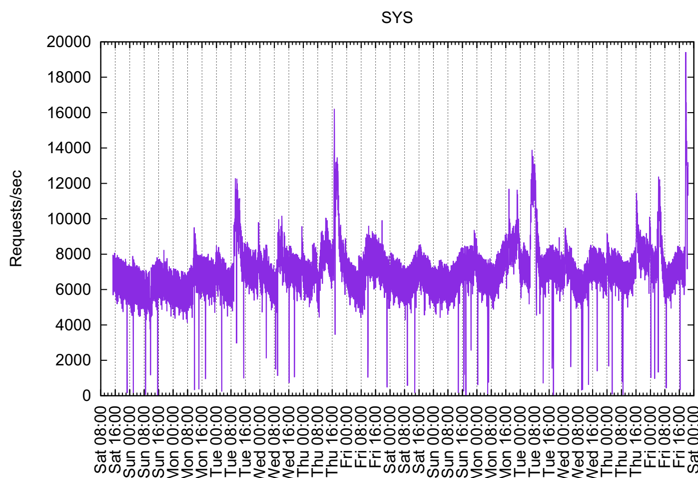
Pool	APP	VAR	SYS	USR	ETC
Hit rate	92.9%	93.7%	98.7%	98.2%	81.4%

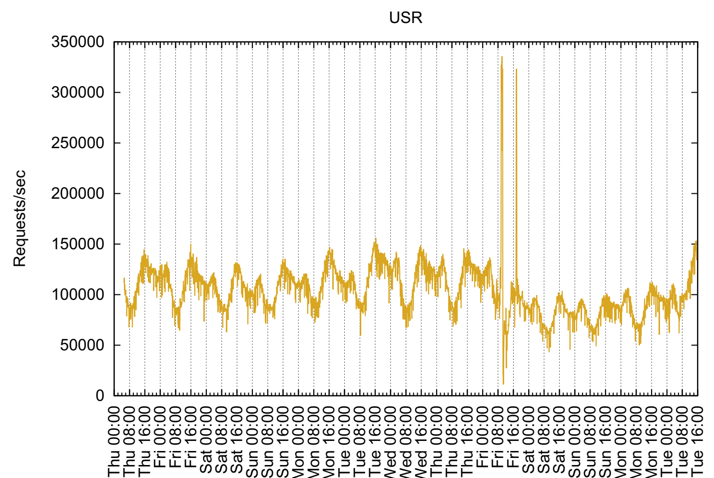
4.1 Hit Rates over Time

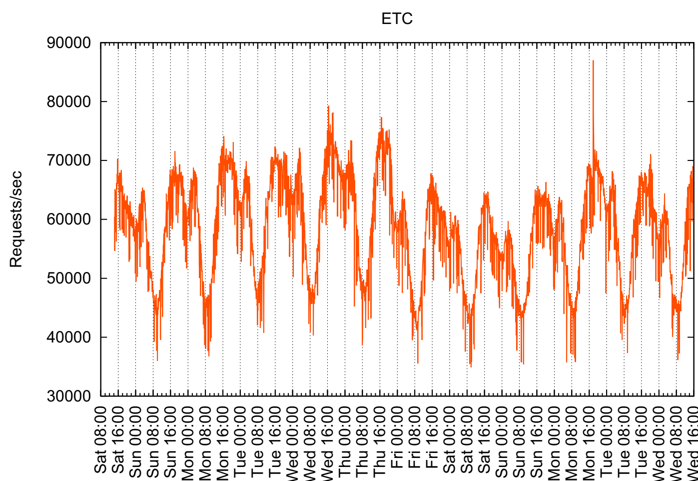
When looking at how hit rates vary over time (Fig. 4), almost all traces show diurnal variance, within a small band of a few percentage points. USR's plot is curious: it appears to be monotonically increasing (with diurnal undulation). This behavior stems from the usage model for USR. Recall











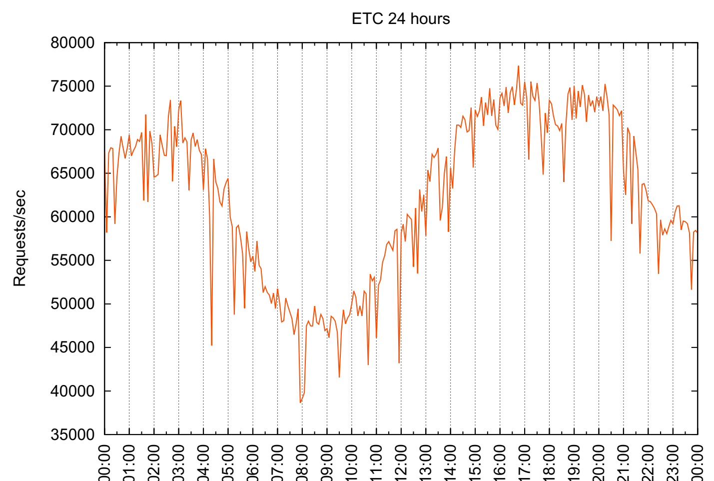


Figure 3: Request rates at different dates and times of day, Coordinated Universal Time (UTC). Each data point counts the total number of requests in the preceding second. Except for USR and VAR, different traces were collected in different times. The last plot zooms in on a 24-hour period from the ETC trace for greater detail.

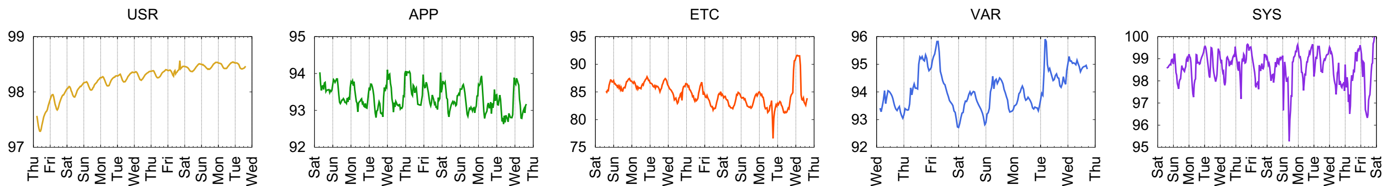


Figure 4: GET hit rates over time for all pools (days start at midnight UTC).

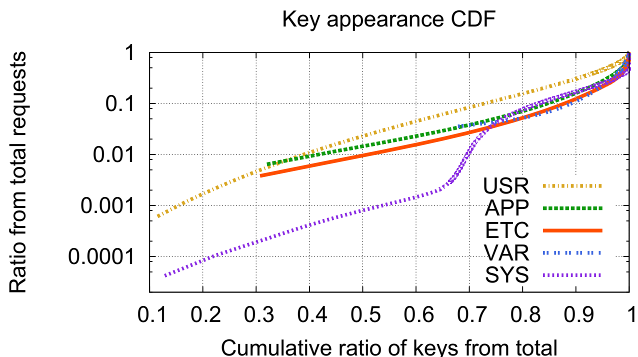


Figure 5: CDFs of key appearances, depicting how many keys account for how many requests, in relative terms. Keys are ranked from least popular to most popular.

from Sec. 2.2 that USR is sized large enough to minimize the number of misses—in other words, to contain almost all possible keys. When a USR Memcached server starts, it contains no data and misses on all requests. But over time, as clients add values to it while the pressure to evict is nonexistent, hit rates climb upwards. Thus, USR's transient hit rate is correlated not only with time of day, but primarily with the server's uptime, reaching 99.8% after several weeks.

Like USR, SYS has a relatively bounded data domain, so it can easily be sized to keep hit rates high and stable. But unlike the other four workloads, SYS does not react directly to user load, so its performance is less cyclical and regular.

4.2 Locality Metrics

This section looks at three ways to measure locality in GET requests: (1) how often and how much some keys repeat in requests; (2) the amount of unique keys and how it varies over time; and (3) reuse period, as a measure of temporal locality.

These metrics, unlike hit rates, are an inherent property of the request stream of each pool; changing the server's hardware or server count will not affect them. Consequently, this data could provide insights toward the workload, in isolation of implementation choices.

4.2.1 Repeating Keys

We start by looking at the distribution of key repeats (Fig. 5). All workloads exhibit the expected long-tail distributions, with a small percentage of keys appearing in most of the requests, and most keys repeating only a handful of times. So, for example, 50% of ETC's keys occur in only 1% of all requests, meaning they do not repeat many times, while a few popular keys repeat in millions of requests per day. This high concentration of repeating keys provides the justification for caching them in the first place.

All curves are remarkably similar, except for SYS's, which has two distinct sections. The first, up to about 65% of the keys, represents keys that are repeated infrequently—conceivably those that are retrieved when one or more clients start up and fill their local cache. The second part, representing the last 25% of keys and more than 90% of the requests, may account for the normal SYS scenario, when a value is added or updated in the cache and all the clients retrieve it.

4.2.2 Locality over Time

It is also interesting to examine how key uniqueness varies over time by counting how many keys *do not* repeat in close time proximity (Fig. 6). To interpret this data, note that a lower percentage indicates that fewer keys are unique, and therefore suggests a higher hit rate. Indeed, note that the diurnal dips correspond to increases in hit rates in Fig. 4.

An immediately apparent property is that in any given pool, this percentage remains relatively constant over time—especially with hour-long bins, with only small diurnal variations and few spikes. Data for 5-minute bins are naturally noisier, but even here most samples remain confined to a narrow range. This suggests that different pools have not only different traffic patterns, but also different caching properties that can benefit from different tuning, justifying the choice to segregate workloads to pools.

Each pool exhibits its characteristic locality band and average. SYS's low average rate of 3.3% unique keys per hour, for example, suggests that different clients request roughly the same service information. In contrast, USR's much higher average rate of 34.6% unique keys per hour, suggests that the per-user data it represents spans a much more disparate range . Generally, we would assume that lower bands translate to higher overall hit rates, all other things being equal. This turns out not to be the case. In fact, the Pearson correlation coefficient between average unique key ratios with 60-minute bins (taken from Fig. 6) and the average hit rates (Table 2) is negative as expected, but small: -0.097. Indeed not all other things are equal, as discussed in Sec. 4.1.

Comparing 5-minute bins to hour-long bins reveals that unique keys in the former appear in significantly higher concentrations than in the latter. This implies a rapid rate of decay in interest in most keys. But does this rate continue to drop very fast over a longer time window? The next section sets out to answer this question.

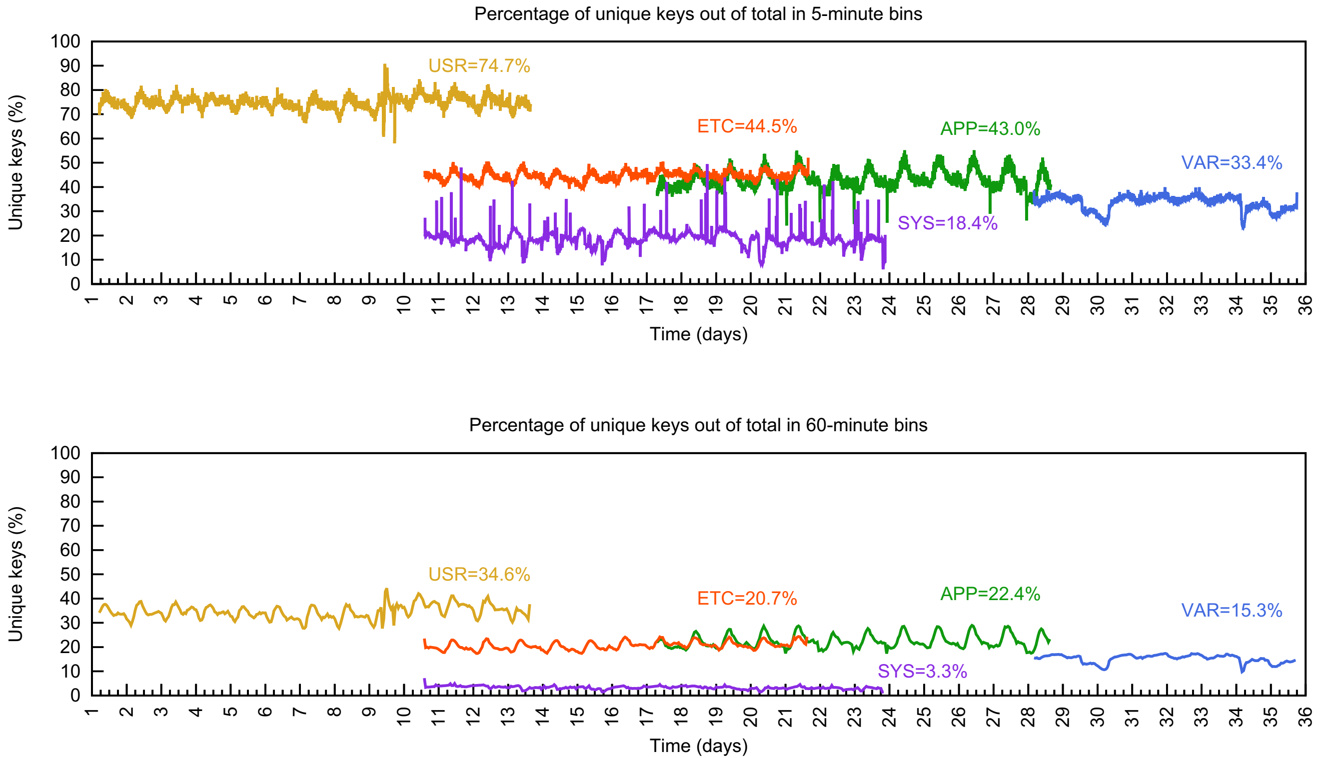


Figure 6: Ratio of unique keys over time. Each data point on the top (bottom) plot shows how many unique keys were requested in the preceding 5 (60) minutes, as percentage of all keys. The label for each pool, at the top right corner of the data, also includes the average ratio throughout the entire pool's trace.

4.2.3 *Temporal Locality: Reuse Period*

Temporal locality refers to how often a key is re-accessed. One metric to quantify temporal locality of any given key is the reuse period—the time between consecutive accesses to the key. Fig. 7 counts all key accesses in the five traces, and bins them according to the time duration from the previous key's access. Unique keys (those that do not repeat at all within the trace period) are excluded from this count.

The answer to the question from the previous section is therefore positive: count of accesses in each reuse period continues to decay quickly after the first hour. For the ETC trace, for example, 88.5% of the keys are reused within an hour, but only 4% more within two, and within six hours, 96.4% of all nonunique keys have already repeated. It continues to decay at a slower rate. This access behavior suggests a pattern for Facebook's users as well, with some users visiting the site more frequently than others and reusing the keys associated with their accounts. Another interesting sub-pattern occurs every day. Note the periodic peaks on even 24 hours in four of the five traces, especially in the VAR pool that is associated with browser usage. These peaks suggest that a noteworthy number of users log in to the site at approximately the same time of day each time. Once more, these increased-locality indications also correspond to increased hit rates in Fig. 4.

As in the previous section, the SYS pool stands out. It does not show the same 24-hour periodicity, because its keys

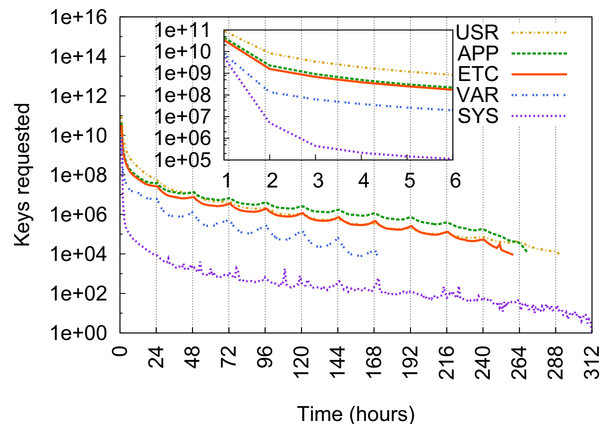


Figure 7: Reuse period histogram per pool. Each hour-long bin n counts keys that were first requested n hours after their latest appearance. The inset zooms in on the five hours after the first.

relate to servers and services, not users. It also decays precipitously compared to the others. As in Sec. 4.2.1, we find that since its data are cached locally by clients, it is likely that most of SYS's GET requests represent data that are newly available, updated or expired from the client cache; these are then requested by many clients concurrently. This would explain why 99.9% of GET requests are repeated within an hour of the first key access. Later, such keys would be cached locally and accessed rarely, perhaps when a newly added client needs to fill its own cache.

Nevertheless, there is still value in reuse period to predict hit rates. Since all pools have sufficient memory for over an hour of fresh data, the percentage of keys reused within an hour correlates positively with the overall hit rates in Table 2 (with a Pearson coefficient of 0.17). The correlation is stronger—with a coefficient of 0.44—if we omit USR and SYS, which have atypical cache behavior (minimum evictions in the former and local caching in the latter).

4.3 Case Study: ETC Hit Rates

We turn our attention to ETC's hit/miss rates, because frequent misses can noticeably hurt user experience. At this point, one might expect ETC's hit rate to exceed the 96% 6-hour key-reuse rate, since it is provisioned with more than enough RAM to store the fresh data of the preceding 6 hours. Unfortunately, this is not the case, and the observed hit rate is significantly lower at 81%. To understand why, we analyzed all the misses in the last 24 hours of the trace (Table. 3). The largest number of misses in ETC comes from keys that are accessed for the first time (at least in a 10-day period). This is the long tail of the locality metrics we analyzed before. Sec. 4.2.1 showed that $\approx 50\%$ of ETC's keys are accessed in only 1% of requests, and therefore benefit little or not at all from a demand-filled cache. The many deletions in the cache also hinder the cache's efficacy. ETC is a very diverse pool with many applications, some with limited reusability. But the other half of the keys that show up in 99% of the requests are so popular (some repeating millions of times) that Memcached can satisfy over 4 in 5 requests to the ETC pool.

Table 3: Miss categories in last 24 hours of the ETC trace. Compulsory misses count GETs with no matching SET in the preceding 10 days (meaning, for all practical purposes, new keys to the cache). Invalidation misses count GETs preceded by a matching DELETE request. Eviction misses count all other missing GETs.

Miss category	Compulsory	Invalidation	Eviction
Ratio of misses	70%	8%	22%

5. STATISTICAL MODELING

This section describes the salient workload characteristics of the ETC trace using simple distribution models. The ETC trace was selected because it is both the most representative of large-scale, general-purpose KV stores, and the easiest to model, since it is not distorted by the idiosyncratic aberrations of application-specific pools. We also think that its mixed workload is easier to generalize to other general-purpose caches with a heterogeneous mix of requests and sizes. The more Facebook-specific workloads, such as USR or SYS, may be interesting as edge cases, but probably not so much as models for synthetic workloads.

The functional models presented here prioritize parsimonious characterization over fidelity. As such, they obviously do not capture all the nuances of the trace, such as its bursty nature or the inclusion of one-off events. But barring access to the actual trace, they can serve the community as a better basis for synthetic workload generation than assumptions based on guesswork or small-scale logs.

Methodology

We modeled independently the three main performance properties that would enable simple emulation of this trace: key sizes, value sizes, and inter-arrival rates. The rate and ratio between GET/SET/DELETE requests can be derived from Sec. 3.1. For cache analysis, additional properties can be gleaned from Sec. 4.

To justify the assumption that the three properties are independent, we picked a sample of $1,000,000$ consecutive requests and measured the Pearson coefficient between each pair of variables. The pairwise correlations, as shown in Table 4, are indeed very low.

Table 4: Pearson correlation coefficient between each two pair of modeled variables.

Variable pair	Correlation
Inter-arrival gap $\leftrightarrow$ Key size	-0.0111
Inter-arrival gap $\leftrightarrow$ Value size	0.0065
Key size $\leftrightarrow$ Value size	-0.0286

We created functional models by fitting various distributions (such as Weibull, Gamma, Extreme Value, Normal, etc.) to each data set and choosing the distribution that minimizes the Kolmogorov-Smirnov distance. All our data resemble power-law distributions, and fit the selected models quite well, with the exception of a handful of points (see Fig. 8). To deal with these outliers and improve the fit, we removed these points as necessary and modeled the remaining samples. The few removed points are tabulated separately as a histogram, so a more accurate synthetic workload of the entire trace should combine the functional model with the short list of value-frequency outliers.

Key-Size Distribution

We found the model that best fits key sizes in bytes (with a Kolmogorov-Smirnov distance of 10.5) to be Generalized Extreme Value distribution with parameters $\mu = 30.7984$, $\sigma = 8.20449$, $k = 0.078688$. We have verified that these parameters remain fairly constant, regardless of time of day.

Value-Size Distribution

We found the model that best fits value sizes in bytes (with a Kolmogorov-Smirnov distance of 10.5), starting from 15 bytes, to be Generalized Pareto with parameters $\theta = 0$, $\sigma = 214.476$, $k = 0.348238$ (this distribution is also independent of the time of day). The first 15 values of length and probabilities can be modeled separately as a discrete probability distribution whose values are given in Table 5.

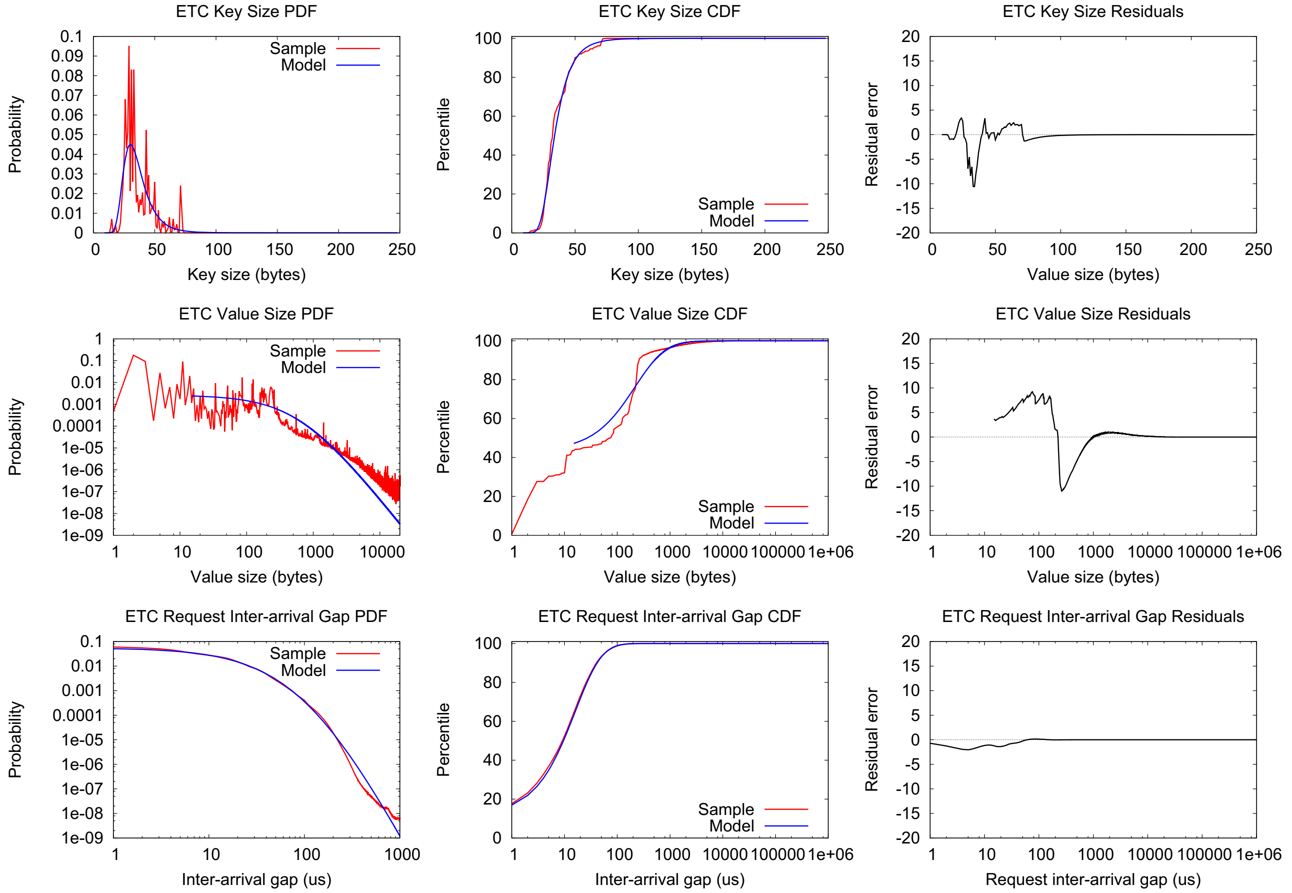


Figure 8: PDF (left), CDF (middle), and CDF residuals (right) plots for the distribution of ETC's key size (top), value size (center). and inter-arrival gap (bottom). Note that some axes are logarithmic, and that PDF plots limit the X-axis to area of interest for greater detail.

Table 5: Probability distribution for first few value lengths, in bytes.

Value size	Probability
0	0.00536
1	0.00047
2	0.17820
3	0.09239
4	0.00018
5	0.02740
6	0.00065
7	0.00606
8	0.00023
9	0.00837
10	0.00837
11	0.08989
12	0.00092
13	0.00326
14	0.01980

Inter-arrival Rate Distribution

We found the model that best describes the time gap in microseconds between consecutive received requests (with a Kolmogorov-Smirnov distance of 2.0) to be Generalized Pareto with parameters $\theta = 0$, $\sigma = 16.0292$, $k = 0.154971$, starting from the second value.

One excluded point from this model is the first value, representing a gap of $0\,\mu sec$ (in other words, multiple requests at the same microsecond time slot), with a probability of 0.1159. This is likely an artifact of our measurement granularity and aggregation by the network stack, and not of concurrent requests, since they are all serialized by the single networking interface.

In addition, the model is most accurate up to about a gap of $1000\,\mu sec$. But the total number of sampled points not covered by this model (i.e., those requests that arrive more than $1msec$ after the previous request) represents less than 0.002% of the total samples and their residual error is negligible.

Unlike the previous two distributions, inter-arrival rate—the reciprocal function of offered load—is highly dependent on time of day, as evident in Fig. 3. For those wishing to capture this diurnal variation, this complete-trace model may be too coarse. To refine this distribution, we divided the

Table 6: Hourly distributions for inter-arrival gap. The columns represent (in order): start time of each hourly bin (in UTC), the two Generalized Pareto parameters (with $\theta = 0$), the fraction of samples under $1\,\mu s$ gap , and the Kolmogorov-Smirnov distance of the fit.

Time	σ	k	$< 1\,\mu s$	KS
0:00	16.2868	0.155280	0.1158	2.18
1:00	15.8937	0.141368	0.1170	2.14
2:00	15.6345	0.137579	0.1174	2.09
3:00	15.7003	0.142382	0.1174	2.16
4:00	16.3231	0.160706	0.1176	2.32
5:00	17.5157	0.181278	0.1162	2.52
6:00	18.6748	0.196885	0.1146	2.64
7:00	19.5114	0.202396	0.1144	2.64
8:00	20.2050	0.201637	0.1123	2.58
9:00	20.2915	0.193764	0.1116	2.46
10:00	19.5577	0.178386	0.1122	2.35
11:00	18.2294	0.161636	0.1130	2.17
12:00	17.1879	0.140461	0.1138	2.00
13:00	16.2159	0.119242	0.1146	1.88
14:00	15.6716	0.104535	0.1152	1.76
15:00	15.2904	0.094286	0.1144	1.72
16:00	15.2033	0.096963	0.1136	1.72
17:00	14.9533	0.098510	0.1140	1.74
18:00	15.1381	0.096155	0.1128	1.67
19:00	15.3210	0.094156	0.1129	1.65
20:00	15.3848	0.100365	0.1128	1.68
21:00	15.7502	0.111921	0.1127	1.80
22:00	16.0205	0.131946	0.1129	1.96
23:00	16.3238	0.147258	0.1148	2.14

raw data into 24 hourly bins and modeled each separately. Fortunately, they all fit a Generalized Pareto distribution with $\theta = 0$ rather well. The remaining two parameters are distributed over time in Table. 6.

6. DISCUSSION

One pertinent question is, what are the factors that affect and predict hit rates? Since all hosts have the same amount of RAM, we should be able to easily explain the relative differences between different traces using the data we gathered so far on locality. But as Sec. 4 discusses, hit rates do not actually correlate very well with most locality metrics, but rather, correlates inversely with the size of the pool (compare Tables 1 and2). Does correlation imply causation in this case?

Probably not. A more likely explanation invokes a third, related parameter: the size of the application domain. Both in USR's and SYS's cases, these sizes are more or less capped, and the bound is small enough that a limited number of servers can cover virtually the entire domain, so locality no longer plays a factor. On the other extreme, ETC has a varying and growing number of applications using it, some with unbounded data. If any single application grows enough in importance to require a certain quality of service, and has the size limitations to enable this quality, given enough servers, then it is separated out of ETC to its own pool. So the applications that end up using ETC are precisely those that cannot or need not benefit from hit-rate guarantees.

Nevertheless, improving hit rates is important for these applications, or we would not need a cache in the first place. One way to improve ETC's hit rates, at least in theory, is to increase the total amount of RAM (or servers) in the pool so that we can keep a longer history. But in practice, beyond a couple of days' worth of history, the number of keys that would benefit from the longer memory is vanishingly small, as Fig. 7 shows. And of course, adding hardware adds cost.

A more fruitful direction may be to focus on the cache replacement policy. Several past studies demonstrated replacement policies with reduced eviction misses, compared to LRU, such as LIRS [19]. Table 3 puts an upper limit on the number of eviction misses that can be eliminated, at around 22%, meaning that eviction policy changes could improve hit rates by another $0.22 \times (1 - 0.814) = 4.1\%$. This may sound modest, but it represents over 120 million GET requests per day per server, with noticeable impact on service latency. Moreover, the current cache replacement scheme and its implementation are suboptimal when it comes to multithreaded performance [9], with its global lock protecting both hash table and slab LRUs. We therefore perceive great potential in alternative replacement policies, not only for better hit rates but also for better performance.

Another interesting question is whether we should optimize Memcached for hit rates or byte hit rates. To answer it, we estimated the penalty of each miss in the ETC workload, by measuring the duration between the missing GET and the subsequent SET that reinstates the value (presumably, the duration represents the time cost to recalculate the value, and is already highly optimized, emphasizing the importance of improved hit rates). We found that it is roughly proportional to the value size, so whether we fill the cache with few large items or many small items of the same aggregate size should not affect recalculation time much. On the other hand, frequent misses do noticeably increase the load on the back-end servers and hurt the user experience, which explains why this cache does not prioritize byte hit rate. Memcached optimizes for small values, because they are by far the most common values. It may even be worthwhile to investigate not caching large objects at all, to increase overall hit rates.

7. RELATED WORK

To the best of our knowledge, this is the first detailed description of a large-scale KV-store workload. Nevertheless, there are a number of related studies on other caching systems that can shed light on the relevance of this work and its methodology.

The design and implementation of any storage or caching system must be optimized for its workload to be effective. Accordingly, there is a large body of work on the collection, analysis, and characterization of the workloads on storage systems, including enterprise computing environments [2, 20, 21] and high-performance computing environments [11, 22, 30]. The observations can be of great importance to system design, engineering, and tuning. For example, in a study on file system workloads for large-scale scientific computing applications, Wang et. al. collected and analyzed file accesses on an 800-node cluster running the Lustre file system at Lawrence Livermore National Laboratory [30]. One of their findings is that in some workloads, small requests account for more than 90% of all requests, but almost all data are accessed by large requests. In a study on file sys-

tem workloads across different environments, Roselli et. al. found that even small caches can produce a high hit rate, but larger caches would have diminishing returns [28], similar to our conclusions on the ETC workload (Sec. 4.3).

In the work describing Facebook's photo storage system [8], the authors presented statistics of I/O requests for the photos, which exhibit clear diurnal patterns, consistent with our observations in this paper (Sec. 3.3).

Web caches are widely deployed as a caching infrastructure for speeding up Internet access. Their workloads have been collected and analyzed in Web servers [6, 25], proxies [5, 10, 16], and clients [7, 12]. In a study of requests received by Web servers, Arlitt and Williamson found that 80% of requested documents are smaller than $\approx 10KB$. However, requests to these documents generate only 26% of data bytes retrieved from the server [6]. This finding is consistent with the distribution we describe in Sec. 3.2

In an analysis of traces of client-side requests, Cunha et. al. show that many characteristics of Web use can be modeled using power-law distributions, including the distribution of document sizes, the popularity of documents, the distribution of user requests for documents, and the number of references to documents as a power law of their overall popularity rank (Zipf's law) [12]. Our modeling work on the ETC trace (Sec. 5) also shows power-law distributions in most request properties.

In light of the increasing popularity and deployment of KV-stores, several schemes were proposed to improve their performance, energy efficiency, and cost effectiveness [4, 9, 15, 26, 29]. Absent well-publicized workload traces, in particular large-scale production traces, many works used hypothetical or synthetic workloads [29]. For example, to evaluate SILT, a KV-store design that constructs a three-level store hierarchy for storage on flash memory with a memory based index, the authors assumed a workload of 10% PUT and 90% GET requests using $20B$ keys and $100B$ values, as well as a workload of 50% PUT and 50% GET requests for $64B$ KV pairs [23]. Andersen et. al. used queries of constant size ($256B$ keys and $1KB$ values) in the evaluation of FAWN, a KV-store designed for nodes consisting of low-power embedded CPUs and small amounts of flash storage [4]. In the evaluation of CLAM, a KV-store design that places both hash table and data items on flash, the authors used synthetic workloads that generate keys from a random distribution and a number of artificial workload mixes [3]. There are also some studies that used real workloads in KV-store evaluations. In two works on flash-based KV store-design, Debnath et. al. adopted workloads from online multi-player gaming and a storage de-duplication tool from Microsoft [13, 14]. Amazon's production workload was used to evaluate its Dynamo KV store, Dynamo [15]. However, these papers did not specifically disclose the workload characteristics.

Finally, there are multiple studies offering analytical models of observed, large-scale workloads. Of those, a good survey of the methods is presented in Lublin's and Feitelson's analysis of supercomputer workloads [17, 24].

8. CONCLUSION AND FUTURE WORK

This paper presented a dizzying number of views into a very large data set. Together, these views tell a coherent story of five different Memcached workloads at Facebook. We have ETC, the largest and most heterogeneous of the five. It has many keys that are requested millions of times a day, and yet its average hit rate is only 81.4% because half of its keys are accessed infrequently, and because a few large values take up a disproportionate amount of the storage. We have APP, which represents a single application and consequently has more uniform objects: 90% of them have roughly the same size. It also mirrors the interest of Facebook's users in specific popular objects, as evidenced in load spikes that are accompanied by improved locality metrics. We have VAR, a transient store for non-persistent performance data. It has three times as many writes as reads and 70% of its keys occur only once. But its 94% hit rate provides noticeable improvements to the user experience. We have USR, which is more like a RAM-based store for immutable two-byte values than a cache. It may not be the best fit for Memcached, but its overall data size is small enough that even a few Memcached servers can deliver a hit rate of 98.2%. And finally, we have SYS, another RAM-based storage that exhibits unique behavior, because its clients already cache its data. They only access SYS when new data or new clients show up, resulting in a low request rate and a nearly bimodal distribution of temporal locality: either a key is accessed many times in a short period, or virtually not at all.

This study has already answered pertinent questions to improve Facebook's Memcached usage. For example, Fig. 7 shows the relatively marginal benefit of significantly increasing the cache size for the ETC pool. As another example, the analysis in Sec. 6 demonstrated both the importance of increasing Memcached's hit rate, especially on larger data, as well as the upper bound on the potential increase.

The data presented here can also be used as a basis for new studies on key-value stores. We have also provided a simple analytical model of ETC's performance metrics to enable synthetic generation of more representative workloads. The treatment of workload modeling and synthetic load generation in this paper only scratches the surface of possibility, and deserves its own focus in a following publication. We plan to focus on this area and model the remaining workload parameters for ETC (such as key reuse), and other workloads as well. With these models, we would like to create representative synthetic load generators, and share those with the community.

We would also like to see improvements in the memory allocation model so that more room is saved for items in high demand. Areas of investigation include an adaptive slab allocation, using no slabs at all, or using prediction of item locality based on the analysis in this study.

Finally, we are looking into replacing Memcached's replacement policy. LRU is not optimal for all workloads, and can be quite slow. We have already started prototyping alternative replacement schemes, and the initial results are encouraging.

Acknowledgements

We would like to thank Marc Kwiatkowski for spearheading this project and Mohan Srinivasan for helping with the kernel module. We are also grateful for the valuable feedback provided by the following: Goranka Bjedov, Rajiv Krishnamurthy, Rajesh Nishtala, Jay Parikh, and Balaji Prabhakar.

9. REFERENCES

[1] http://voldemort-project.com.

[2] Ahmad, I. Easy and efficient disk I/O workload characterization in VMware ESX server. In *Proceedings of IEEE International Symposium on Workload Characterization* (Sept. 2007).

[3] Anand, A., Muthukrishnan, C., Kappes, S., Akella, A., and Nath, S. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation* (Apr. 2010).

[4] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. FAWN: a fast array of wimpy nodes. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (Big Sky, Montana, 2009), ACM, pp. 1–14.

[5] Arlitt, M., Friedrich, R., and Jin, T. Workload characterization of a web proxy in a cable modem environment. *ACM SIGMETRICS - Performance Evaluation Review 27* (1999), 25–36.

[6] Arlitt, M. F., and Williamson, C. L. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking 5* (October 1997), 631–645.

[7] Barford, P., Bestavros, A., Bradley, A., and Crovella, M. Changes in web client access patterns. In *World Wide Web Journal, Special Issue on Characterization and Performance Evaluation* (1999).

[8] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (Oct. 2010).

[9] Berezecki, M., Frachtenberg, E., Paleczny, M., and Steele, K. Many-core key-value store. In *Proceedings of the Second International Green Computing Conference* (Orlando, FL, Aug. 2011).

[10] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the 18th Annual IEEE International Conference on Computer Communications* (1999).

[11] Carns, P., Latham, R., Ross, R., Kamil Iskra, S. L., and Riley, K. 24/7 characterization of petascale I/O workloads. In *Proceedings of the 4th Workshop on Interfaces and Architectures for Scientific Data Storage* (Nov. 2009).

[12] Cunha, C. R., Bestavros, A., and Crovella, M. E. Characteristics of WWW client-based traces. In *Technical Report TR-95-010, Boston University Department of Computer Science,* (July 1995).

[13] Debnath, B. K., Sengupta, S., and Li, J. Flashstore: High throughput persistent key-value store. *Proceedings of 36th International Conference on Very Large Data Bases (VLDB) 3*, 2 (2010).

[14] Debnath, B. K., Sengupta, S., and Li, J. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the Annual ACM SIGMOD Conference* (June 2010), pp. 25–36.

[15] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (Stevenson, WA, 2007), pp. 205–220.

[16] Duska, B. M., Marwood, D., and Feeley, M. J. The measured access characteristics of world-wide web client proxy caches. In *Proceedings of USENIX Symposium of Internet Technologies and Systems* (Dec. 1997).

[17] Feitelson, D. G. Workload modeling for performance evaluation. In *Performance Evaluation of Complex Systems: Techniques and Tools*, M. C. Calzarossa and S. Tucci, Eds., vol. 2459 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 2002, pp. 114–141. www.cs.huji.ac.il/~feit/papers/WorkloadModel02chap.ps.gz.

[18] Fitzpatrick, B. Distributed caching with memcached. *Linux Journal*, 124 (Aug. 2004), 72–78. www.linuxjournal.com/article/7451?page=0,0.

[19] Jiang, S., and Zhang, X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (2002), SIGMETRICS'02, ACM, pp. 31–42.

[20] Kavalanekar, S., Worthington, B., Zhang, Q., and Sharda, V. Characterization of storage workload traces from production windows servers. In *Proceedings of IEEE International Symposium on Workload Characterization* (Sept. 2008).

[21] Keeton, K., Alistair Veitch, D. O., and Wilkes, J. I/O characterization of commercial workloads. In *Proceedings of the 3rd Workshop on Computer Architecture Evaluation using Commercial Workloads* (Jan. 2000).

[22] Kim, Y., Gunasekaran, R., Shipman, G. M., Dillow, D. A., Zhang, Z., and Settlemyer, B. W. Workload characterization of a leadership class storage cluster. In *Proceedings of Petascale Data Storage Workshop* (Nov. 2010).

[23] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. Silt: A memory-eficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (Oct. 2011).

[24] Lublin, U., and Feitelson, D. G. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing 63*, 11 (Nov. 2003), 1105–1122. www.cs.huji.ac.il/~feit/papers/Rigid01TR.ps.gz.

[25] Manley, S., and Seltzer, M. Web facts and fantasy. In *Proceedings of USENIX Symposium on Internet Technologies and Systems* (Dec. 1997).

[26] Petrovic, J. Using Memcached for data distribution in industrial environment. In *Proceedings of the Third International Conference on Systems* (Washington, DC, 2008), IEEE Computer Society, pp. 368–372.

[27] Reddi, V. J., Lee, B. C., Chilimbi, T., and Vaid, K. Web search using mobile cores: Quantifying and mitigating the price of efficiency. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)* (June 2010), ACM. portal.acm.org/citation.cfm?id=1815961.1816002.

[28] Roselli, D., Lorch, J. R., and Anderson, T. E. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference* (June 2000).

[29] Vasudevan, V. R. *Energy-Efficient Data-intensive Computing with a Fast Array of Wimpy Nodes.* PhD thesis, Carnegie Mellon University, Oct. 2011.

[30] Wang, F., Xin, Q., Hong, B., Miller, E. L., Long, D. D. E., Brandt, S. A., and McLarty, T. T. File system workload analysis for large scientific computing applications. In *Proceedings of 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies* (Apr. 2004).

[31] Zhao, H. HipHop for PHP: Move fast. https://developers.facebook.com/blog/post/358/, Feb. 2010.

Ding C, Xiang X, Bao B *et al.* Performance metrics and models for shared cache. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 29(4): 692–712 July 2014. DOI 10.1007/s11390-014-1460-7

Performance Metrics and Models for Shared Cache

Chen Ding[1] (丁　晨), Xiaoya Xiang[1] (向晓娅), Bin Bao[1] (包　斌), Hao Luo[1] (罗　昊), Ying-Wei Luo[2] (罗英伟) and Xiao-Lin Wang[2] (汪小林)

[1]*Department of Computer Science, University of Rochester, Rochester, NY 14627-0226, U.S.A.*

[2]*School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China*

E-mail: cding@cs.rochester.edu; {sappleing, bin.bao}@gmail.com; hluo@cs.rochester.edu; {lyw, wxl}@pku.edu.cn

Received March 1, 2014; revised May 14, 2014.

Abstract Performance metrics and models are prerequisites for scientific understanding and optimization. This paper introduces a new footprint-based theory and reviews the research in the past four decades leading to the new theory. The review groups the past work into metrics and their models in particular those of the reuse distance, metrics conversion, models of shared cache, performance and optimization, and other related techniques.

Keywords memory performance metric, cache sharing, reuse distance

1 Introduction

Computing is ubiquitous in science, engineering, business, and everyday life. Most of today's applications, whether for cloud, desktop, or handheld, run on multicore processors. As a result, they interact with peer programs. It is beneficial to minimize the negative interaction. The benefit is important not just for good performance but also for stable performance, not just for parallel code but also for sequential applications running in parallel.

This paper surveys the theories and techniques to measure and improve program interaction on multicore processors. A program is either a sequential application or a parallel application being treated as a single party in interaction. Here we assume that programs do not share data or computation, but they share the hardware host. We call it a *solo-run* if a program runs by itself on a machine and a *co-run* if multiple programs run in parallel.

Cache sharing is a primary cause of co-run interference. Modern applications take most of their time to access memory, and most memory accesses — over 99% typically — happen in cache. A commodity system today has 2 to 8 processors (sockets), 2 to 6 physical cores per processor, and 2 to 4 hyperthreaded logical cores per physical core. Nearly a hundred programs can run together in parallel.

Partitioned cache solves the interference problem via program isolation. However, cache partitioning is wasteful when only one program is running and inefficient when co-run programs share data. Current multicore processors use a mix of private and shared cache. For example, Intel Nehalem has 256 K L2 cache per core and 4 MB to 8 MB L3 cache shared by all cores. IBM Power 7 has 8 cores, with 256 KB L2 cache per core and 32 MB L3 shared by all cores.

Depending on which CPU they are using, programs interact in different ways. Physical cores have private caches at the first and second levels but share the last level cache. Logical cores share the caches at all levels. Different processors do not share the caches. However, they share the memory bandwidth, and the demand of memory bandwidth depends entirely on the performance of the cache. In addition, some caching policies, e.g., inclusive cache on Intel machines, may induce indirect interaction, where a program may lose data in its private cache due to the data access by another program in the shared cache.

The advent of cache sharing the 2000s is reminiscent of the middle 1960s when time sharing was invented. Since then, the problem of memory management has been well studied and solved, and modern operating systems routinely manage memory for a large number of programs. However, the problem of cache sharing is more complex.

Survey

The work is partially supported by the National Natural Science Foundation of China (NSFC) under Grant No. 61232008, the NSFC Joint Research Fund for Overseas Chinese Scholars and Scholars in Hong Kong and Macao under Grant No. 61328201, the National Science Foundation of USA under Contract Nos. CNS-1319617, CCF-1116104, CCF-0963759, an IBM CAS Faculty Fellowship and a research grant from Huawei. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding organizations.

Xiang has graduated and is now working at Twitter Inc. Bao has graduated and is now working at Qualcomm Inc.

©2014 Springer Science + Business Media, LLC & Science Press, China

Cache is managed by hardware, not the operating system. Cache has multiple levels and varying mixes of exclusivity and sharing. Events of cache accesses and replacements are orders of magnitude more frequent than memory access and paging. A single program may access cache a billion times a second and can wipe out the entire content of the cache in less than a millisecond. The intensity multiplies when more programs are run in parallel. Furthermore, the size of cache is fixed on a given machine. One cannot get online and buy more cache as one can with memory.

Cache interference is asymmetrical, non-linear, and circular. The asymmetry was shown experimentally by Zhang *et al.*[1] at Rochester and confirmed by later studies. In a pair-run experiment we conducted using Zhang's setup. One program becomes 85% slower, while its partner is only 15% slower. The interference changes from program to program. The effect depends not as much on how many programs are running as on which programs are running. Finally, the effect is circular. As a program affects its peers, it is also affected by them.

The solution to these problems requires a special theory called the theory of locality. Locality is a basic property of a computing system. Denning[2] defined locality as "a concept that a program favors a subset of its segments during extended intervals (phases)." There is a difference between the data that a program has and the data that the program is actively using. The "active" data is a subset, which Denning[3] called *the working set*.

Performance depends on how fast a computer system provides access to the active data subset. The access time of the other data is irrelevant. Locality analysis is therefore a prerequisite to memory design, for the oft quoted reason "we cannot improve what we cannot measure." In this article, we review the metrics for measuring and techniques for improving performance in shared cache.

2 Footprint Theory of Locality

2.1 Footprint

As a locality metric, the *footprint* measures the amount of active data usage. Given a program execution, we extract the data accesses as a linear sequence of memory addresses or object IDs. The sequence is called an access trace or an address string. A window is a sub-sequence of consecutive accesses. The length of a window is measured by time, either logically based on the number of accesses in the window or physically based on the time when the first and the last accesses were made.

Given a window, the footprint is the amount of data accessed in the window, i.e., the size of the "active" data. For an execution, the footprint is defined for each window length as the *average* footprint of all windows of that length. In a dynamic execution, the data usage may change in different length windows and in different windows of the same length. The footprint shows the change over all window lengths. For each length, it shows the average footprint, which is a single, unique value.

For example, consider three data blocks a, b, c. Fig.1 shows two patterns of data accesses. One has a stack access pattern, where the data block last accessed is first reused. The other has a streaming pattern, where the blocks are traversed in the same order. The footprints are shown for all length-3 windows, four in each trace. The footprint of a trace is the average. For length-3 windows, the footprint, $fp(3)$, is 2.5 in the stack trace and 3 in the streaming trace. Therefore, the streaming access has a greater data activity for that window length. The complete footprint is defined for all window lengths and would count in the amount of data access in all windows of all lengths.

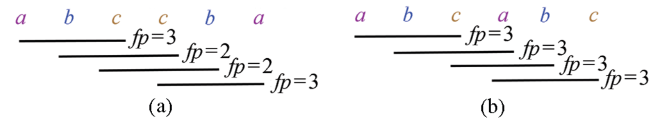


Fig.1. Amount of data accessed in length-3 windows in two access traces: (a) stack accesses and (b) streaming accesses. The footprint of a trace is the average amount. It is defined for each window length. When the length is 3, the footprint, $fp(3)$, is 2.5 in the stack trace and 3 in the streaming trace.

In practice, the footprint is too numerous to enumerate. The number of time windows is quadratic to the length of the trace①. Assuming a program running for 10 seconds on a 3 GHz processor, we have 3E10 CPU cycles in the execution and 4.5E20 distinct windows.

Brock *et al.*[4] described program analysis as a Big Data problem, and showed the scale of the problem by the number of time windows in an execution. Fig.2 shows that as the length of execution increases from 1 second to 1 month, the number of CPU cycles (n) ranges from 3E9 to 2E15, and the number of distinct execution windows $\binom{n}{2}$ from 4.5E18 to 5.8E29, that is, from 4 sextillion to over a half nonillion.

As a dynamic analysis problem, the scale quickly reaches the size of any static problem. As a comparison, the figure shows the radius of the Milky Way in centimeters, 48 sextillion, and the radius of the observable universe, 44 octillion.

①If the trace length is n, the number of windows (and hence footprints) is $\binom{n}{2} + n = \frac{n\times(n+1)}{2}$ or $O(n^2)$ asymptotically.

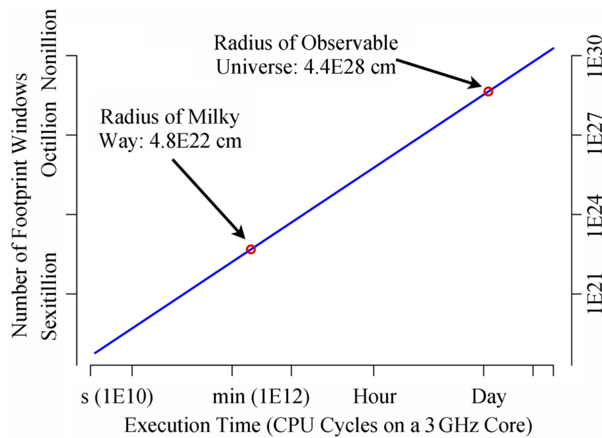


Fig.2. Scale of the problem shown by the number of footprint windows in a program execution, compared to the size of a galaxy and the universe. Reproduced from [4].

For system design, it may not be very useful to consider very large windows, since caching decisions are usually based on information on the recent execution. For programming, however, it is necessary to analyze the full execution to find opportunities of global optimization. This is shown by Zhong *et al.* in whole-program locality analysis, which analyzes the full length of reuse distances to see how it changes with the input[5], and in affinity-based data layout, which groups structure fields based on the distribution of long reuse distances[6].

The purpose of a footprint theory is to overcome the enormity of the analysis problem, characterize the active data usage in all windows, and make it useful for system analysis and optimization.

2.2 Footprint Theory

For locality analysis, the basic unit of information is a data access, and the basic relation is a data reuse. The theory of locality is concerned with the fundamental properties of data accesses and reuses, just as the graph theory is with nodes and their links.

The footprint theory consists of a set of formal definitions, algorithms, and properties based on the concept of the footprint. This subsection introduces the four components of the theory and their supporting techniques, based on the material published in a series of papers[7-12].

Footprint Measurement. The enormous scale of all-window analysis is tackled by a series of three algorithms. Each is two orders of magnitude more efficient than the previous one.

• Footprint distribution analysis, which enumerates all $O(n^2)$ footprints in $O(n \log m)$ time, where n is the length of the trace and m the maximal footprint.

• Average footprint analysis, which reduces the cost to linear time $O(n)$ by computing the average without enumerating all footprints.

• Footprint sampling, which samples limited-size windows and further reduces the cost.

The distribution analysis is the first algorithm to measure the all-window footprint. As it actually enumerates all footprints, it finds the largest, smallest, median, average, and any percentile footprint for each window length. However, the cost is sometimes thousands of times slowdown compared to the speed of the original program.

The second algorithm computes just the average footprint, and the cost is reduced from a thousand times slowdown to about 20 times. Being a linear time algorithm, it is scalable in that the cost increases proportionally to the length of the program execution.

The cache on a real machine has a finite size, so an analysis does not have to consider windows whose footprint is greater than the cache size. In addition, the behavior of a long running program tends to repeat itself. Furthermore, on modern processors, the analysis can be carried out on a separate core in parallel with the analyzed execution. Footprint sampling specializes and parallelizes the analysis for a specific machine and program. The average cost is reduced to 0.5% of the running time of the unmodified execution.

The algorithmic development attains immense gains in both computational complexity and implementation efficiency. As the baseline, the distribution analysis is the first viable solution for precise all-window analysis. The second and the third algorithm each improves efficiency by another order of magnitude, eventually making it fast enough for real-time analysis. This has a beneficial impact elsewhere, because the footprint can be used to compute other locality metrics, as we will see in the third part of the footprint theory.

Composability. A locality metric is *composable* if the metric of a co-run can be computed from the metric of solo-runs. If co-run programs do not share data, the footprint is composable. Let the average footprint of a program be $prog.fp(x)$ for window length x. If we have k programs $prog_1, prog_2, \ldots, prog_k$ actively sharing the cache, the aggregate footprint is the sum of the individual footprints.

$$corun.fp(x) = \sum_{i=1}^{k} prog_i.fp(x).$$

In comparison, the miss ratio is not composable. We will prove it later in Subsection 3.6.3. Intuitively, the co-run miss ratio will change compared to the solo-run miss ratio, since each program has now a fraction instead of the whole cache. The change in miss ratio, as mentioned earlier, is asymmetrical, non-linear, and

affected by circular feedback. As a result, we cannot directly add the solo-run miss ratio to compute the co-run miss ratio, as we can with the footprint.

Another locality metric is reuse distance. Reuse distance does not depend on cache parameters, but as we will explain in Subsection 3.2, neither is it composable.

As mentioned earlier, we can measure the average as well as the distribution of footprints. The average footprint is immediately composable. The distribution, although composable, requires a convolution which is expensive to compute and difficult to visualize. In the following, the term "footprint" means the average footprint.

The next question is whether the footprint composability can help in analyzing the miss ratio and other locality metrics in shared cache. This is solved in the third part of the new theory.

Locality Metrics Conversion. Locality has different measurements, just like temperature can be measured in different scales, Celsius or Fahrenheit. For locality, the two most common metrics are miss ratio for hardware design and reuse distance for program optimization.

Central to a locality theory is the conversion between different metrics. The footprint theory shows that the footprint is convertible with a number of other metrics. Let $mr(c)$ be the miss ratio for cache size c. It can be computed from the footprint using the following formula[10]:

$$mr(c) = \frac{fp(x + \Delta x) - fp(x)}{\Delta x},$$

where $c = fp(x)$. If these are continuous functions, we would say that the miss ratio is the derivative of the footprint.

The higher order mathematics implies mathematical properties. Since the derived metric, the miss ratio, is non-decreasing, the source metric, the footprint, must be not just non-decreasing, but also concave. Indeed, the monotonicity and concavity were proved in two successive papers[9-10].

The conversion is reversible. If we have the miss ratios of all cache sizes, we can reverse the formula and compute the average footprint. The reverse process is the analog of integration for a discrete function.

Combining footprint composition and metrics conversion, we can see immediately that if the co-run miss ratio (miss ratio seen by the shared cache) can be computed from the aggregate footprint. Fig.3 shows the derivation by adding the individual footprints and then converting the sum into the co-run miss ratio.

Since the conversion formula is reversible, we can switch between the footprint and the miss ratio and compose the latter indirectly through the former. First, we compute the individual footprint from the individual miss ratios (of all cache sizes). Then we add the individual footprints and finally compute the co-run miss ratio in the shared cache (of all sizes). Fig.3 shows this type of deduction and others that are made possible by composition and conversion. In particular, the figure shows how to compose another locality metric, the reuse distance. We use the terms private reuse distance (PRD) and concurrent reuse distance (CRD), as introduced by [13-14].

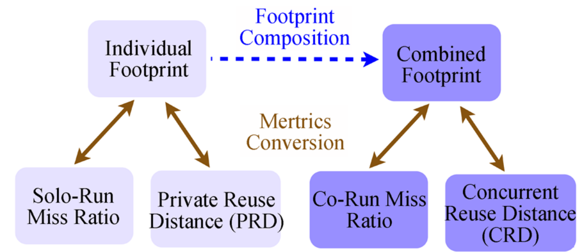


Fig.3. Joint use of two theoretical properties: composition (dotted line) and conversion (solid lines).

The solution of composition raises the problem of decomposition. The co-run miss ratio does not tell us the contribution from each program. To see the individual effects, we need more elaborate models.

Composable Locality Models. We say a model is composable if the co-run result can be computed from solo-run results, not just for the co-run group as a whole, but also for the co-run effect on each individual program. In other words, a composable model must be both composable and decomposable.

As a composable metric, the footprint has the following useful traits:

• *Machine Independent.* The analysis is based on data accesses, not cache misses. It takes a single pass to analyze a trace for all cache sizes, and the result is not affected by program instrumentation. In comparison, it is inescapable for direct measurement to be affected by instrumentation.

• *Clean-Room Statistics.* The footprint of one program can be measured in a co-run environment, unperturbed by other programs. The clean-room effect solves the chicken-or-egg problem of direct measurement: the behavior of one program depends on its peer, but the peer behavior in turn depends on itself.

• *Peer Independent.* The footprint of a program is independent of co-run peers. The analysis of cache sharing does not require actual cache sharing. The co-run effect is computed rather than measured.

• *Statically Composable.* There are 2^P co-run combinations for P programs. The footprint model can predict the interference in these 2^P runs by testing P single-program runs. For the P sequential runs, we can

choose to run them one by one or some of them in parallel to increase speed. The composition is static if there is no actual co-run; otherwise we say the composition is dynamic. Here dynamic composition means parallel testing, while static composition does not need parallel testing at all.

To compute the co-run effect on each individual program, this dissertation describes three models. The models solve the decomposition problem as a composition problem: how one program is affected by its peers.

• *Composition by Reuse Distance and Footprint.* Variations of this model were invented by Thiebaut and Stone[15] and Suh *et al.*[16] for time-sharing systems (time-switched cache sharing) and Chandra *et al.*[17] for multicore (continuous cache sharing). These studies estimated the footprint since there was no feasible ways to measure it. After the invention of the fast measurement, the cost of the model became limited by the time required for reuse distance measurement[9-10].

• *Composition by Footprint Only.* The second model converts the footprint into reuse distance, so it no longer needs to measure the reuse distance and can be hundreds of times faster[10].

• *Composition by Program Pressure and Sensitivity.* The last model is as fast as the second model but more intuitive and easier to use. It characterizes the behavior of a program by two factors, pressure and sensitivity. The two can be visualized as two curves. Performance composition is as simple as looking up related values on the two curves[12].

The composable models provide answers to a number of long-standing questions about shared cache, including a machine independent way to compare programs by their shared cache behavior, the correlation between a program's cache interference and its miss ratio, and the performance of cache sharing compared with cache partitioning[12].

These models are theoretical, and they are appealing partly due to the generality. The footprint is defined on a program trace without knowing co-run peers or machine parameters (other than having shared cache). There are many sources of error due to the fact that the basic models do not consider the effect of cache associativity, program phase behavior, the time dilation due to interference, the filtering effect in a multi-level cache hierarchy, and the impact of the prefetcher. A theory must be validated to be practically relevant. The past studies have used experiments on real systems to evaluate the theoretical models and compare their predictions with actual miss counts measured by hardware counters[8-10,12]. They also showed extensions of the models to consider time dilation[8,12], cache associativity, and program phases[10].

3 Locality Theory from 1968

Locality was started as an observation that programs do not use all their data at all times[2]. After decades of research, it has been developed into an important scientific field. At its foundation are locality metrics, so the concept and its effect can be measured. Among the basic problems are the measurement speed and accuracy of these metrics.

3.1 Miss Ratio and Execution Time

The metric of miss ratio was first used by Belady[18] to find out how often individual policies caused page faults. It was challenging at that time to measure page traces and simulate the various policies on them. Today, the hardware performance counters on modern machines enable a tool to measure program speed and count cache misses in real time with little cost. The performance of a single program or a group of programs can be observed directly. However, direct observation has difficulties in characterizing the locality cleanly due to dependences on the observation environment. These dependences include:

• *Machine Dependence.* Different machines have different memory hierarchies and processors, so we cannot compare the locality in different programs entirely based on their performance.

• *Instrumentation Dependence.* The analysis code itself consumes processor and cache resources. It may not be possible to completely separate the effect of the instrumentation.

• *Peer Dependence.* It is unknown how the performance has changed due to cache sharing. It would have required another test on an unloaded system. It is also unknown how the performance will change if the peer programs change.

The effect of cache on performance is often disruptive. This phenomenon was first discussed by Denning[19] and stated as the *thrashing*, which happens when the sum of the working sets exceeds the available memory. Chilimbi once compared the phenomenon to strolling leisurely until suddenly falling over a cliff②. The danger is greater in a shared environment. As more programs are added, the combined working set grows. When it exceeds the size of the shared cache, sharp performance drops would ensue. Being peer and machine dependent, direct testing cannot foresee a pending calamity. What is worse, it cannot even tell whether a

②Trishul Chilimbi made this analogy in a presentation in 2002[20].

given parallel mix is efficient or not without testing them individually first.

3.2 Reuse Distance

The most common metric in program characterization is the reuse distance. For each memory access in a trace, the reuse distance is the number of distinct data elements accessed between this and the previous access to the same data. Mattson *et al.* first defined the concept (to model the performance of an LRU stack) and called it the LRU stack distance[21]. LRU is a cache management method that favors recently used data. Recognizing it as a measure of recency, Jiang and Zhang[22] called the metric the inter-reference recency (IRR).

For example, the reuse distance shows the locality of stack access and streaming access traces in Fig.4. When a block is first accessed, the reuse distance is infinite. When the block is reused, the reuse distance is the number of distinct blocks accessed between the previous access and the reuse. The reuse would miss in (fully associative LRU) cache if and only if its reuse distance is greater than the cache size. The figure shows that the stack trace can reuse the data in cache when the cache size is less than 3 but the streaming trace cannot.

Reuse Distance	∞	∞	∞	1	2	3	∞	∞	∞	3	3	3
	a	*b*	*c*	*c*	*b*	*a*	*a*	*b*	*c*	*a*	*b*	*c*

Fig.4. The locality of two traces, stack accesses on the left and streaming accesses on the right, measured by the reuse distance of each memory reference. An access is a miss in fully associative LRU cache if and only if its reuse distance is greater than the cache size.

The reuse distance quantifies the locality of every memory access. The locality of a program, or a loop or a function inside it, is the collection of all its reuse distances. The collective result can be represented as a distribution. It is called a *locality signature*[5] and *locality profile*[14].

3.2.1 Relation with Cache Performance

In the absence of cache sharing, the capacity miss ratio can be written as the fraction of the reuse distance that exceeds the cache size[21]. Let the test program be A and cache size be C; we have

$$P(\text{capacity miss by} A) = P(A\text{'s reuse distance} > C). \quad (1)$$

The reuse distance is machine independent but can give the capacity miss ratio for cache of all sizes, as the formula shows. The locality signature can be viewed as a discrete probability density function, showing the probability of a memory access having a certain degree of locality. The miss ratio is then the probability function, showing the probability of the access being a miss for a given cache size. A probabilistic adjustment invented by Smith can estimate the effect of cache conflicts in set-associative cache[23-25]. Combining the reuse distance and the Smith formula, we can compute the miss ratio in the cache of all sizes.

Miss Ratio Curve (*MRC*). The miss ratio curve (MRC) shows the miss ratio of all its cache sizes as a discrete function. It is easy to visualize and show directly the trade-off between performance and cache size. For fully associative LRU cache, the miss ratio curve is equivalent to the reuse distance distribution, as the preceding formula shows. The problem is equivalent in theory to the argument whether it is measuring the miss ratio curve or the reuse distance. In practice, the miss ratio curve is defined for only practical cache sizes, i.e., powers of two between some range, e.g., 32 KB and 8 MB. The reuse distance has the full range between 1 and the size of program data.

The full range of reuse distance represents the complete temporal locality. The miss ratio curve is a projection of the full information on a subset of cache sizes. The two would be equivalent if the miss ratio is defined for all cache sizes between 1 and infinity. The unbounded size of the representation is necessary, as shown by the theoretical result of Snir and Yu[26] that temporal locality cannot be fully encoded using a bounded number of bits. In the following, we review the prior work on both the reuse distance and the miss ratio curve.

3.2.2 Locality Analysis and Optimization

Reuse distance has found many uses. The locality signature shows how the cache behavior changes with the program input, and the changes can be predicted by whole-program locality analysis[5,25,27], which was used to predict the miss ratio of all inputs and cache sizes[28]. Fang *et al.* modeled locality signature for each memory reference and used it to find critical memory loads and important program paths[27,29]. Marin *et al.*[25] modeled the locality signature at reference, loop, and function levels to predict performance across different computer architectures. Beyls and D'Hollander[30-31] built a program tuning tool *SLO*, which identifies the cause of long distance reuses and gives improvement suggestions for restructuring the code. In addition to cache misses, reuse distance has been used to analyze the response time in server systems[32] and the usage pattern in web reference streams[33]. Zhong *et al.*[5] classified these and other uses of reuse distance as "Five Dimen-

sions of Locality" and reviewed the analysis techniques for program input, code, data, execution phase, and program interaction.

Reuse distance provides a common foundation to model program behavior, predict machine performance, and guide program optimization. Locality analysis and profiling are to infer, measure, and decompose reuse distances, and locality optimization is to shorten long reuse distances. The analysis and the optimization are free of machine, instrumentation, and peer dependences. The downside, however, is the complexity of measuring reuse distance.

3.2.3 Direct Measurement

Reuse distance is one of the stack distances defined in the seminal paper in 1970 by Mattson *et al.*[21] The stack algorithm in the paper needs $O(nm)$ time to profile a trace with n accesses to m distinct data. The efficiency has been steadily improved over the past four decades. In 1975, Bennett and Kruskal[34] organized the trace as a tree and reduced the cost to $O(n \log n)$. In 1980, Olken[35] made the tree compact and reduced the cost further to $O(n \log m)$.

The Olken algorithm has been the most efficient asymptotic solution (for full reuse distance measurement) until 2003, when Ding and Zhong gave an approximation algorithm[5,36]. The approximation guarantees a relative precision, e.g., 99%, and takes $O(n \log\log m)$ time, which is effectively linear to n for any practical data size m. Zhong *et al.* also gave an algorithm that guarantees a constant error bound and does not reduce the asymptotic cost[37]. In an independent implementation, Schuff *et al.*[38] reported that the average cost of the $O(n \log\log m)$ method is as high as several thousand times slowdown.

Kim *et al.*[39] gave a linear-time algorithm to measure the miss ratio for a fixed number of cache sizes, which may be used to approximate reuse distance.

There are practical improvements to the Olken algorithm. *Cheetah* implemented the Olken algorithm using a splay-tree[40]. It became part of the widely used SimpleScalar simulator[41]. Almasi *et al.*[42] used a different tree representation to further improve the efficiency. A greater efficiency can be obtained through sampling and parallelization (Subsections 3.2.6 and 3.2.7).

Zhong *et al.* gave a lower bound result showing that the space cost of precise measurement is at least $O(n \log n)$, indicating that reuse distance is fundamentally a harder problem than streaming, i.e., counting the number of 1's in a sliding window, which can be done using $O(n)$ space[5].

3.2.4 Approximation by Reuse Time

While the reuse distance counts the number of distinct memory accesses, the reuse time counts all accesses. It is simply the difference in logical time between the previous access and the current reuse and can be measured quickly in $O(n)$ time. The working set theory uses the reuse time (inter-reference gap) to compute the time-window miss rate[43]. If we take time-window miss rate as an approximation of the LRU miss rate, we may say that the working set theory is the first approximation technique.

Two series of more recent studies have used the reuse time to compute the reuse distance. The first is StatCache and StatStack by Hagersten and his students[44-47],③, and the second is time-based locality approximation[48-50]. For brevity, we name the latter technique after its lead author and call it the *Shen conversion*.

Berg and Hagersten solved the following equation for the miss ratio R[44]. Let N be the length of the trace, $h(t)$ be the number of accesses whose reuse time is t, and $f(k)$ be the probability that a cache block is replaced after k misses. The cache is assumed to have random replacement, so $f(k) = 1 - (1 - \frac{1}{C})^k$, for cache with C blocks. The total number of misses can be computed in two ways, and they should be equal:

$$NR = \sum_{t=1} Nh(t)ft(R).$$

StatCache solves the implicit equation for the miss ratio R using numerical methods.

In the Shen conversion[48,51], the key measure is the interval access probability $p(\Delta)$, which is the probability of a randomly chosen datum v being accessed during a time interval Δ. For a reuse at time distance Δ, below is the probability that its reuse distance is k:

$$p(k, \Delta) = \binom{N}{k} p(\Delta)^k (1 - p(\Delta))^{N-k}.$$

The formula computes the probability for k distinct data items to appear in a Δ-long interval. It assumes a binomial distribution given the interval access probability $p(\Delta)$, which is computed as

$$p(\Delta) = \sum_{t=1}^{\Delta} \sum_{\delta=t+1}^{T} \frac{1}{N-1} p_T(\delta), \quad (2)$$

where $p_t = \frac{h(t)}{N}$ is the probability that an access has the time distance t. The derivation for $p(\Delta)$ can be found in a technical report[51].

③Berg and Hagersten used the term reuse distance for what we mean by reuse time[44].

The two statistical techniques are successful in predicting performance. StatCache was used to model first private cache[44-45] and then shared cache[46-47]. The Shen conversion was used first for sequential code[48-49] and then multi-threaded code[50,52].

Although both using statistical analysis, StatCache and the Shen conversion are fundamentally different: one models the random cache, and the other the LRU cache. Next we explore the difference between random and LRU modeling in greater depth.

3.2.5 *Random vs LRU*

Any statistical analysis of locality invariably makes some assumptions about randomness. We examine three such assumptions.

The first is random access to a cache set, which means that a data access can happen at any cache set with equal probability. The Smith formula uses the assumption to calculate the contention in a cache set and the effect of cache associativity[23].

The second is random cache replacement, which means that a miss may evict any cache block with equal probability. Under the assumption of random replacement, the lifetime of a block in cache is binomially distributed over the number of cache misses. Not knowing the miss rate, StatCache uses the relation to compute the miss rate from the reuse time[44]. Knowing the miss rate, West *et al.* computed the cache occupancy[53]. Fedorova *et al.* devised a fair scheduling policy based on the assumption that a set of applications divide the cache equally if they had the same miss rate[54].

Since real cache does not use random replacement, the accuracy of the assumption needs to be examined. For cache occupancy, West *et al.* compared the prediction with the actual measurement (through cache simulation) and found that the prediction is accurate for caches using random replacement but less so for caches using LRU[53].

Random has two other differences from LRU. One is well known, which is that the random replacement cache is fully associative by definition. The other is less recognized, which is that the cache performance is not deterministic as the replacement decisions change randomly every time a program is run. Fortunately, the problem is recently solved. Zhou gave an ingenious algorithm to compute the average miss ratio in a single pass, without having to simulate multiple times to compute the average[55].

The way to model LRU is using reuse distance. Knowing the reuse distance, the Smith formula uses it to model the LRU replacement within a cache set[23]. Not knowing the reuse distance, the Shen conversion needs a way to compute it[48,51]. It assumes a third type of randomness — in a time window, each data block is uniformly randomly accessed. By computing the reuse distance, the Shen conversion models LRU rather than random cache replacement.

Cache models can be divided by the replacement policy: LRU or random. There is a second dimension to compare them: the metrics used to measure window-based locality. For random replacement, we want to know the number of misses in a time window. There is a (trivial) linear relation between the miss count and the window length. For LRU, we want to know the footprint in a window. The relation is non-linear, and it is the main source of complexity in the Shen conversion in particular the derivation of the interval access probability.

Cache models use two types of window-based locality: the miss count and the footprint. The miss count is linear but cache size dependent. In comparison, the footprint is non-linear but cache size independent. For example, StatCache has to solve its equation for every cache size, while the Shen conversion produces the reuse distance and the miss ratio for all cache sizes. The past solutions represent different trade-offs between modeling simplicity and power. With the footprint theory, we have a new option, which is to compute the reuse distance using the footprint, which we can measure as accurately as we can with the miss count.

The three modeling methods, StatCache, the Shen conversion, and the footprint conversion, are not guaranteed to always give the correct reuse distance. Indeed, a precise linear-time solution is unlikely given the lower bound result in Zhong *et al.*[5] Among the three, the footprint theory is unique in formulating the condition for correctness, which is the reuse hypothesis[10].

3.2.6 *Sampling Analysis*

Sampling is usually effective to reduce the cost of profiling. Choosing a low sampling rate may reduce the amount of profiling work by factors of hundreds or thousands. In program analysis, bursty tracing is widely used, where the execution alternates between short sampling periods and long hibernation periods[20,56-57]. During hibernation, the execution happens in the original code and has no analysis overhead.

Locality sampling, however, is tricker. Locality is about the time of data reuse, but the time is unknown until the access actually happens. The uncertainty has two consequences. First, the length of the sampling period cannot be bounded if it is to cover a sampled data reuse pair. Second, the analyzer has to keep examining every data access. Complete hibernation is effectively impossible.

The problem of locality sampling is addressed by a series of studies, including the publicly available SLO tool[30], continuous program optimization[58], bursty

reuse distance sampling[59], and multicore reuse distance analysis[38]. Sampling can drastically reduce the cost if sampled windows accurately reflect the behavior of other windows[45-47].

SLO has been developed by Beyls and D'Hollander[30]. It instruments a program to skip every k accesses and take the next address as a sample. A bounded number of samples are kept in a sample reservoir — hence the name reservoir sampling. To capture the reuse, SLO checks each access to see if it reuses some sample data in the reservoir. The instrumentation code is carefully engineered in GCC to have just two conditional statements for each memory access (one for address and the other for counter checking). Reservoir sampling reduces the time overhead from 1000-fold slow-down to only a factor of 5 and the space overhead to within 250 MB extra memory. The sampling accuracy is 90% with 95% confidence. The accuracy is measured in reuse time, not reuse distance or miss rate.

To accurately measure reuse distance, a record must be kept to count the number of distinct data that appeared in a reuse window. Zhong and Chang[59] developed the bursty reuse distance sampling, which divides a program execution into sampling and hibernation periods. In the sampling period, the counting uses a tree structure and costs $O(\log \log M)$ per access. If a reuse window extends beyond a sampling period into the subsequent hibernation period, counting uses a hash-table, which reduces the cost to $O(1)$ per access. Multicore reuse distance analysis by Schuff *et al.*[38] uses a similar scheme for analyzing multi-threaded code. Its fast mode improves over hibernation by omitting the hash-table access at times when no samples are being tracked. Both methods track reuse distance accurately.

StatCache by Berg and Hagersten[45] is based on unbiased uniform sampling. After a data sample is selected, StatCache puts the page under the OS protection (at page granularity) to capture the next access to the same datum. It uses the hardware counters to measure the time distance until the reuse. OS protection is limited by the page granularity. Two other systems, developed by Cascaval *et al.*[58] and Tam *et al.*[60], use the special support on IBM processors to trap accesses to specific data addresses. To reduce the cost, these methods use a small number of samples. Cascaval *et al.*[58] used the Hellinger Affinity Kernel to infer the accuracy of sampling. Tam *et al.*[60] predicted the miss rate curve in real time.

3.2.7 Parallel Analysis

Schuff *et al.*[38] combined sampling and parallel analysis for parallel code on multicore. At the IPDPS conference in 2012, three groups of researchers reported that they made the analysis of even sequential programs many times faster with parallel algorithms. Niu *et al.*[61] parallelized the analysis to run on a computer cluster, while Cui *et al.*[62] and Gupta *et al.*[63] parallelized it for GPU.

Unlike the reuse distance, the footprint can be easily sampled and analyzed in parallel using shadow profiling[64-65]. By measuring the footprint and converting it to reuse distance, we have shown the equivalent of parallel sampling analysis for reuse distance, which can be done in near real-time, with just 0.5% visible cost on average[10]. We note that the accuracy of footprint conversion is conditional[10], but direct (parallel) measurements are always accurate.

3.2.8 Compiler Analysis

Reuse distance can be analyzed statically for scientific code. Cascaval and Padua[66] used the dependence analysis[67], and Beyls and D'Hollander[68] defined *reuse distance equations* and used the Omega solver[69]. While they analyzed conventional loops, Chauhan and Shei[70] analyzed MATLAB scripts using dependence analysis. Unlike profiling whose results are usually input specific, static analysis can identify and model the effect of program parameters. Beyls and D'Hollander[68] used the reuse distance equations for cache hint insertion, in particular, conditional hints, where the caching decision is based on program run-time parameters. Shen *et al.*[71] used static and lightweight reuse analysis in the IBM compiler for array regrouping and structure splitting.

Using the static reuse distance analysis and the footprint theory, Bao and Ding demonstrated a compiler technique for analyzing the program footprint and discussed the potential use in peer-aware program optimization[72-73]. In [72], they used the tiled matrix multiply (Fig.5) as an example to show the reuse distance computed at the source level (Table 1). They also

```
for (jj = 0; jj < N; jj = jj + B_j)
  for (kk = 0; kk < N; kk = kk + B_k)
    for (i = 0; i < N; i = i + 1)
      for (j = jj; j < min(jj + B_j, N); j = j + 1)
        for (k = kk; k < min(kk + B_k, N); k = k + 1)
          C[i][j] = beta × C[i][j] + alpha × A[i][k] × B[k][j];
```

Fig.5. Loop nest of tiled matrix multiply.

Table 1. Reuse Distance as a Function of the Loop Bounds

Loop	Array	Reuse Distance (Bytes)
k	$C[i][j]$	8×3
j	$A[i][k]$	$8 \times 1 + 8 \times B_k + 8 \times B_k$
i	$B[k][j]$	$8 \times B_j + 8 \times B_k + 8 \times B_k \times B_j$
kk	$C[i][j]$	$8 \times N \times B_j + 8 \times N \times B_k + 8 \times B_k \times B_j$
jj	$A[i][k]$	$8 \times N \times B_j + 8 \times N \times N + 8 \times N \times B_j$

showed the use of the conversion theory (Subsection 2.2) to compute the miss ratio curve and a measure of shared-cache friendliness called the fill time.

3.2.9 Domain-Specific Modeling

To model graph algorithms, Yuan *et al.*[74] defined the notion *vertex distance* and used statistical analysis to derive the reuse distance. The study examines random graphs and scale-free graphs. It shows the dual benefits of domain-specific analysis. On the one hand, the structure of a graph facilitates locality analysis. On the other hand, locality analysis reveals the relation between the properties of a graph, e.g., edge density, and the efficiency of its computation.

3.2.10 Discussion

Reuse distance is a powerful tool for program analysis. It quantifies the locality of every program instruction. For a single sequential execution, the metric is composable. For example, the composition can happen structurally to show the locality of larger program units such as loops, functions, and the whole program, or it can happen temporally to show program executions as (integer valued) signals.

There are at least two limitations. First, reuse distance is insufficient to analyze program interaction. While programs interact at all times in the shared cache, reuse distance provides locality information for only reuse windows, not all windows. Second, precise reuse distance is still costly to measure. Despite all of the advances in sampling and parallelization, the asymptotic cost is still more than linear. These problems will be addressed indirectly through the study of another locality metric, the footprint.

3.3 Early Footprint

Measuring footprint requires counting distinct data elements, and the result depends on observation windows. The problem has long been studied in measuring various types of reuse distances as discussed before. However, footprint measurement is a more difficult problem than reuse distance measurement. Given a trace of length n, there is only $O(n)$ reuse windows but in total $O(n^2)$ footprint windows. This subsection focuses on the measurement problem, which prior work solved by either selecting a window subset to measure or constructing a model to approximate.

Direct Counting for Subset Windows. Agarwal *et al.*[75] counted the number of cold-start misses for all windows starting from the beginning of a trace (*cumulative cold misses*). Cumulative cold misses, together with warm-start region misses, were used to evaluate cache performance degradation caused by operation system and multiprogramming activity.

The footprint in single-length execution windows can be computed in linear time. On time-shared systems, the window of concern is the scheduling quantum. On these systems, the cached data of one process may be evicted by data brought in by the next process. Thiebaut and Stone computed what is essentially the single-window footprint by dividing a trace by the fixed interval of CPU scheduling quantum and taking the average amount of data access of each quantum[15].

Ding and Chilimbi[7] gave a sampling solution. At each access, it measures the footprint of a window ending at the current access. The length of the measured window is chosen at random.

For an execution of length n, direct counting measures the footprint in $O(n)$ windows. If we use direct counting to estimate all-window footprint, we have a sampling rate $O(\frac{1}{n})$. The sampling rate may be too low to be statistically meaningful, or it may be sufficient in practice. Without a solution for all-window analysis, we would not have a way to evaluate the accuracy of direct counting.

Footprint Equations. Suh *et al.*[16] and Chandra *et al.*[17] used a recursive equation to estimate the footprint. As a window of size w is increased to $w+1$, the change in the footprint depends on whether the new access is a miss. The equation is as follows: consider a random window w_t of size t being played out on some cache of infinite size. As we increase t, the footprint increases with every cache miss. Let $E[w_t]$ be the expected footprint of w_t, and $M(E[w_t])$ be the probability of a miss at the end of w_t. For window size $t+1$, the footprint either increases by an increment of one or stays the same depending on whether $t+1$ access is a cache miss.

$$E[w_{t+1}] = E[w_t](1-M(E[w_t]))+(E[w_t]+1)M(E[w_t]).$$

The term $M(E[w_t])$ requires simulating sub-traces of all size t windows, which is impractical. Suh *et al.*[16] solved it as a differential equation and made the assumption of linear window growth when the range of window sizes under consideration is small. On the other hand, Chandra *et al.*[17] computed the recursive relation bottom up. Neither method can guarantee a bound on the accuracy, i.e., how the estimate may deviate from the actual footprint.

In addition, these approaches produce the average footprint, not the distribution. The distribution can be important. Consider two sets of footprints, A and B. One tenth of A has size $10N$ and the rest has size 0. All of B has size N. A and B have the same average footprint N, but their different distribution can lead

to very different types of cache interference. With the footprint distribution analysis[8], we now have a way to evaluate whether the average footprint produces the same composition results as the footprint distribution.

The past solutions on reuse distance often make similar estimates because the reuse distance is the footprint in a reuse window. These techniques[45,48-50,76] were mentioned in Subsection 3.2. They do not guarantee the precision of the estimation.

3.4 Analytical Models

Instead of measuring the reuse distance or footprint, a mathematical model may be used to characterize the cache performance. Apex-Map uses a parameterized model and a probe program to quickly find the model parameter for a program and a machine[77]. Ibrahim and Strohmaier[78] compared the result of synthetic probing and that of reuse distance profiling, while He *et al.*[79] used a fractal model to estimate the miss rate curve through efficient online analysis.

There was much work earlier on analytical models for memory paging performance. An extensive survey can be found in [2]. Saltzer[80], a designer of the Multics system, gave one simple formula (Subsection 3.6.1). He explained that "Although it is only occasionally that a mathematically tractable model happens to exactly represent the real-world situation, often an approximate model is good enough for many engineering calculations. The challenge ... is to maintain mathematical tractability in the face of obvious flaws and limitations in the range of applicability and yet produce a useful result." Saltzer's formula has been used by Strecker[81] in cache modeling.

Another type of analytical models is the *independent reference model*. Given a program with n pages, each has an independent access probability p that adds to 1, King[82] showed that a steady miss rate exists for fully associative caches managed by LFU, LRU, and FIFO replacement policies. Later studies gave efficient approximation methods for LRU and FIFO[83-84]. Gu and Ding[85] proved a simple relation between random access and the reuse distance distribution (which is uniform). The method of Dan and Towsley[84] can be used to analyze a more general case where data is divided into multiple groups and has different (random access) probabilities. It is a type of composable model.

3.5 Metrics Conversion and Denning's Law

Footprint is a form of working set. The working set theory is the scientific basis as much for memory management as it is for cache management. Denning defined the working set precisely as "the set of distinct pages referred to in a backward window of fixed size T."④ The average footprint for window length T is the average working set size for all size T windows.

A breakthrough in this area is a simple formula discovered by Denning④ and first published in 1972[43]. It shows the relation between the working set size, which is difficult to measure, and the frequency and interval of data reuses, which are easy to measure. The formula converts between two locality metrics. Metrics conversion is at the heart of the science of locality, because it shows that memory behavior and performance are different displays of the same underlying property.

While the proof of Denning and Schwartz[43] depends on idealized conditions in infinitely long executions, subsequent research has shown that the working set theory is accurate and effective in managing physical memory for real applications.

There are three ways to quantify the working set: as a limit value in Denning's original paper[3], as the time-space product defined by Denning and Slutz[86], and as the all-window footprint just defined in Subsection 3.3 (initially in [7]). The equation Denning discovered holds in all three cases. In our 2013 paper[10], we stated it as a law of locality and named it after its discoverer:

Denning's Law of Locality 1. *The working set is the second-order sum of the reuse frequency, and conversely, the reuse frequency is the second-order difference of the working set.*

The footprint theory subsumes the infinitely long case in the original working set theory and proves Denning's law for all executions. It gives a theoretical explanation to the long observed effectiveness of the working set theory in practice.

Easton and Fagin[87] gave another important formula for the conversion between the cold-start and warm-start miss ratios. The authors called it their "recipe". The recipe reveals that the (cold-start) lifetime in cache size C is the sum of the inter-miss times of the (warm) cache for sizes smaller than C. They found that their "estimate was almost always within 10∼15 percent of the directly observed average cold-start miss ratio." They further quoted the analysis of [88] as corroborating evidence. In these studies, as in the work of Denning and Schwartz[43], a program is assumed to be a stationary stochastic process. In the footprint theory, the Easton-Fagin formula can be derived directly, and the theory shows the correctness condition when it is used for finite-length program executions.

④Personal communication, December 17, 2013.

3.6 Locality Models of Shared Cache

3.6.1 Early Models

There are two types of cache sharing: the sharing between multiple time-switched programs, and the sharing between the instruction and data of the same program. Easton and Fagin[87] studied the former, comparing the difference between cold-start and warm-start miss ratios and computing the effect of task interruptions as a weighted average of expected cold-start miss ratios. Thiebaut and Stone[15] defined a precise measure called the *reload transient*. For a departing process, the reload transient is the amount of its cached data lost when it returns after another process is run. To compute the reload transient, Thiebaut and Stone[15] defined *cache footprint*, which is the number of data blocks a program has in cache. Given two programs A, B, the reload transient of A after B is the overlap between their cache footprints.

To compute footprints and their overlap, Thiebaut and Stone[15] assumed that a program has an equal probability of accessing any cache block. The probability is independent and identically distributed. The overlap is then computed from expectations of binomial distributions.

Instead of discrete probabilistic models, Strecker[81] put forward an intuitive notion that a program is a *continuous flow* and fills the cache at the rate that is the product of two probabilities: the chance of a miss and the chance that the miss results in a new location in the cache being filled. A differential equation was constructed since the fill rate is the derivative of the footprint over time. To compute the miss ratio, Strecker[81] used an analytical formula by Saltzer[80]. Saltzer[80] computed the inter-miss time in which he called the *headway* as the number of hits between successive misses.

The second type of cache sharing happens between the instruction and the data of a program. Stone *et al.*[89] investigated whether LRU produces the optimal allocation. Assuming that the miss rate functions for instruction and data are continuous and differentiable, the optimal allocation happens at the points "when miss-rate derivatives are equal"[90]. The miss rate functions, one for instruction and one for data, were modeled instead of measured. The authors showed that LRU is not optimal, but left open a question as to whether there is a bound on how close LRU allocation is to optimal allocation. The footprint theory can be used to compute the effective cache allocation (LRU allocation) among any group of programs.

As a component of the Wisconsin Wind Tunnel (WWT) project, Falsafi and Wood[91] developed a performance model for cache. They used the formulation of Thiebaut and Stone[15] but computed the overlap using a queuing model. In implementation, they measured the cold-start miss rate and used a reverse mapping to estimate the footprint. Since WWT ran the concurrent processes of a parallel program, the instruction code was shared between processes. The sharing was modeled as the shared footprint in the overall process footprint.

Falsafi and Wood[91] revised the terminology of Thiebaut and Stone[15] and redefined the *footprint* as the set of unique data blocks a program accesses. The *projection* of the footprint is the set of data blocks that the program leaves in cache. Viewed in another way, the footprint is the program data in an infinite cache, and the projection is the data in a finite cache. The footprint theory uses their definition of the word *footprint*.

3.6.2 Reuse Distance in Parallel Code

Reuse distance measures the locality of a program directly and does not rely on the assumptions that are necessary for analytical models. In a parallel program, we have two types of reuse distance. One considers only the accesses of a single task, and the other considers the interleaved accesses of all tasks. Using the terminology of Wu and Yeung[13] and Jiang *et al.*[50], we call them private reuse distance (PRD) and concurrent reuse distance (CRD). The new problem in analyzing the parallel locality is the relation between PRD and CRD.

Recent work has studied several solutions. Ding and Chilimbi[76] built models of data sharing and thread interleaving to compose CRD. Jiang *et al.*[50] tackled the composition problem using probabilistic analysis, in particular, the interval access probability based on [48], discussed in Subsection 3.2.

Multicore reuse distance by Schuff *et al.*[38] measures CRD directly using improved algorithms made efficient by sampling and parallelization. For loop-based code, Wu and Yeung gave a scaling model to predict how the reuse distance, both PRD and CRD, changes when the work is divided by a different number of threads[13]. These modeling techniques have found uses in co-scheduling[52] and multicore cache hierarchy design[13-14,92].

3.6.3 Non-Composability of Reuse Distance

A model is composable if the locality of a parallel execution can be computed from the locality of individual tasks. However, the reuse distance is insufficient to build composable models.

We illustrate this limitation by a counter example, first published in [8]. Fig.6 shows three short program traces. Programs A, B have the same set of private reuse distances (PRD). However, when running with a third program C, the pair A, C produces a different set of concurrent reuse distances (CRD) than the pair B, C. Assuming that the cache size is 4, the pair A, C has no capacity miss, but B, C has. The example also shows the same limitation for miss ratio. With identical reuse distances, A, B have the same number of misses in the private cache. But in the shared cache co-running with the same program C, they incur a different number of cache misses.

Fig.6 is a disproof by counterexample. It shows conclusively that PRD is not enough to compute CRD, and the solo-run miss ratio is not enough to compute the co-run miss ratio.

In the example, the reason for the different co-run locality is the different interaction based on the time span of a reuse. Consider the data accesses to a in A, B. They have the same private reuse distance, 2, but very different (logical) reuse times, 3 in A and 7 in B. When co-running with C, the reuse distance is lengthened because of the data accesses by C. Since the reuse in B spans over a longer time, it is affected more by cache sharing. As a result, the concurrent reuse distance for a is 4 in the A, C but 5 in the B, C co-run.

Chandra *et al.*[17] described three models of cache sharing. A simple one is the composition of reuse distance, called (LRU) *stack distance competition* (SDC). Since the model uses the reuse distance as the only input, it would have given the same prediction in our example for A, C and B, C. Therefore, it is a flawed model. A number of earlier studies have reached the same conclusion through experiments[93-95].

3.6.4 Classic Composition Model

Let A, B be two programs that share the same cache but do not share data. The effect of B on the locality of A is:

$$P\,(\text{capacity miss by } A \text{ when running with } B)$$
$$=P(A\text{'s reuse distance} + B\text{'s footprint} \geqslant \text{cache size}).$$

In this model, the cache interference (i.e., CRD) is computed by combining the footprint (i.e., the interference), and the reuse distance, i.e., the per-task locality. Specialized versions of this model were first developed by Suh *et al.*[16] for time-sharing systems and Chandra *et al.*[17] for multicore cache sharing. While Chandra[17] described and evaluated the composition for two programs, Chen and Aamodt[96] improved the accuracy when analyzing more programs with a greater number of cache conflicts. A later study by Jiang *et al.*[52] gives the general form of the classic model not tied to cache parameters such as associativity.

In the work of Suh *et al.*[16] and Chandra *et al.*[17], the footprint equation is iterative (see Subsection 3.3), while in the work of Jiang *et al.*[52], the footprint equation is statistical (see Subsection 3.2). Another footprint equation is the conversion formula by Denning and Schwartz[43]. These equations are not completely constrained, so the solution is not unique and depends on modeling assumptions.

The classic model is not simple as presented in the previous publications. In the work of Chandra *et al.*[17], hardware and program factors were considered together. Xie and Loh[97] noted that the model by Chandra *et al.* "is fairly involved; the large number of complex statistical computations would be very difficult to directly implement in hardware." In addition, the model has a high cost. It was not used in the comparison study of Zhuravlev *et al.*[94], because it was not "computationally fast enough to be used in the robust scheduling algorithm."

There is another weakness in usability. The two inputs, reuse distance and footprint, do not have a simple effect on the composed output. The complexity hinders the use of composable model in practice. As introduced in Subsection 2.2, the footprint theory shows many equivalent methods of composition. The subsection lists two other methods that are faster and easier to use.

3.7 Performance and Optimization

Cache is one of the factors in machine performance. Locality models show the frequency of cache hits and

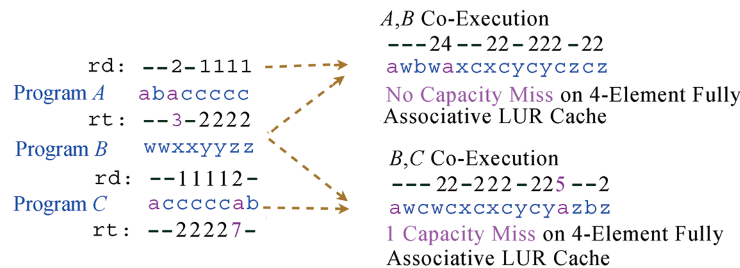


Fig.6. Non-composability of reuse distance. Programs A, B have the same set of reuse distances ("–" means *infty*), but A, C co-run has a different set of reuse distances than B, C co-run does.

misses. For performance analysis, the next question is the combined effect on the execution time, and the ultimate question is the limit to which the performance can be improved.

3.7.1 From Cache Misses to CPU Cycles

The effect of cache on execution time is traditionally given by two metrics, AMP (average miss penalty) and AMAT (average memory access time), which is the number of cycles necessary for respectively, a cache miss and a memory access on average[98].

On modern processors, the timing effect is increasingly complex. A recent analysis was conducted by Sun and Wang[99], who explained that AMAT is affected by the processor techniques for improving instruction-level parallelism, including pipelining, multiple functional units, out-of-order execution, branch prediction, speculation, and by the techniques for improving memory performance, including pipelined, multi-port, multi-bank cache, non-blocking cache, and data prefetching. The increasing complexity motivates the development of new metrics such as APC (access per cycle) studied in their paper[99].

Much of the timing delay is caused by events outside the CPU and the cache, in particular, the memory controller, the memory bus and the DRAM modules.

Zhao *et al.*[100] developed a model of pressure that includes both cache and memory bandwidth sharing using regression analysis to identify a piece-wise linear correlation between the memory latency and the memory bandwidth utilization. The model is not peer specific. The same utilization may be caused by one program or a group of programs. Wang *et al.*[101] gave an event model called DraMon to capture the probability of DRAM hits, misses and conflicts and the effect of contention and concurrency at the level of a DRAM bank. The event model was shown to be more accurate than linear and logarithmic regression[102].

It is important to manage contention and sharing at the memory layer, as shown by two recent techniques, bus-cycle modulation for execution throttling[103] and memory partitioning to reduce bank-level interference[104]. Next we turn the attention back to cache and review the techniques for reducing the cache interference.

3.7.2 Characterization of Interference

Xie and Loh[97] gave an animalistic classification of program interference. Based on the behavior in shared cache, a program belongs to one of the four animal classes. A turtle has little use of shared cache. A rabbit and a sheep both have a low miss rate. A rabbit is sensitive and tends to be affected by co-run peers, but a sheep does not. Both programs have small impacts on others. The last class is a devil, which has a high miss rate and impairs the performance of others but is not affected by others.

Other classifications include coloring of miss intensity, dual metrics of cache partitioning, and utility of cache space to performance. These are reviewed by Xie and Loh[97].

Jiang *et al.*[52] classified programs along two locality dimensions. The *sensitivity* is computed from the classic composition model (Subsection 3.6.4). It shows how a program is affected by others. The *competitiveness* is distinct data blocks per cycle (DPC), which is equivalent to the average footprint. If we divide each locality dimension into two halves, we have four classes, which we may call *locality classes*. Locality classes are not the same as animal classes. For example, a program can be extremely competitive, i.e., devilish, but may also be either sensitive or insensitive. This phenomenon was observed by Zhuravlev *et al.*[94], who showed that "devils were some of the most sensitive applications".

3.7.3 Optimal Co-Scheduling

Given a set of programs, co-scheduling divides them into co-run groups, where each group is run together. The goal is to minimize the interference within these groups, so to maximize resource utilization and co-run throughput. The interference depends mostly on the memory hierarchy, and the effect is non-linear and asymmetric.

While a locality model may predict the cache interference, the impact on performance depends on many other factors including the CPU speed, the effect of prefetching, the available memory bandwidth, and, if a program is I/O intensive, the speed of the disk or the network. Direct testing can most accurately measure the performance interference. Complete testing, however, has an exponential cost, since the number of subsets in an n-element set is 2^n. Note that solo executions are needed to compute the slowdown in group executions.

For pairwise co-runs, the interference can be represented by a complete graph where nodes are programs and edges have weights equal to pair-run slowdowns. Jiang *et al.*[105],⑤ showed that the optimization is min-weight perfect matching, and the problem is NP-hard. They gave an approximation algorithm that produces near-optimal schedules.

The throughput is often not the only goal. Other desirable properties include fairness, i.e., no program is

⑤First published by Jiang *et al.*[106]

penalized disproportionally due to unfair sharing, and quality of service (QoS), i.e., a program must maintain a certain level of performance.

As inputs, an optimal solution requires accurate prediction of co-run degradation. Prior solutions are either locality based (see Subsection 3.6) or performance based (this subsection). It is difficult for them to produce accurate prediction without expensive testing. For co-run miss rates, the footprint gives near real-time prediction, with an accuracy similar to exhaustive testing[10].

3.7.4 Heuristics-Based Co-Scheduling

In symbiotic scheduling (SOS), Snavely and Tullsen[107] used a sampling phase to test a number of possible co-run schedules and select the best one from these samples for the next (symbiosis) phase. They showed that a small number of possible schedules (instead of exhaustive testing) is sufficient to produce good improvements. The system was designed and tested for simultaneous multi-threading. Symbiotic scheduling assumes that program co-run behavior does not vary significantly over time, so the sampling phase is representative of performance in the remaining execution. Testing does not require program instrumentation.

Fedorova *et al.*[54] addressed the problem of performance isolation by suspending a program execution when needed. They gave a cache-fair algorithm to ensure a program runs at least at the speed with fair cache allocation. The technique is based on the assumption that if two programs have the same frequency of cache misses, they have the same amount of data in cache. In locality modeling, the assumption means uniform distribution of the access in cache. While the assumption is not always valid, the model is efficient for use in an OS scheduler to manage cache sharing in real time.

The two techniques are dynamic and do not need off-line profiling. However, on-line analysis may not be accurate and cannot predict interference in other program combinations. Furthermore, non-symbiotic pairing (during sampling) and throttling (for fairness) do not maximize the throughput.

Blagodurov *et al.*[95],⑥ developed the *Pain* classification. The degree of pain that application A suffers while it co-runs with B is affected by A's cache sensitivity, which is computed using the reuse distance profile (PRD), and B's cache intensity, which is measured by the number of last level cache accesses per million instructions. The Pain model is similar to the classic composition model described in Subsection 3.6.4 except that Pain uses the miss frequency rather than the footprint. The choice is partly for efficiency. Other online techniques also use the last-level cache miss rate as cache use intensity[108-109].

Pain is an offline solution. This idea is extended into an online solution called Distributed Intensity (DI), which uses only the miss rate. An application is classified as intensive if it is above the average miss rate and non-intensive otherwise. The scheduler then tries to group high-resource-intensity program(s) with low-resource-intensity program(s) on a multicore to mitigate the conflicts on shared resources[3,18,93-95,110-111].

Cache misses represent only a (small) subset of program accesses. In comparison, the footprint includes the effect of all cache accesses. Furthermore, the miss frequency depends on co-run peers and has the effect of circular feedback, since the peers are affected by the self. The result of counter-based modeling is specific to one grouping situation and may not be usable in other groupings. In comparison, footprint analysis collects "clean-room" statistics, unaffected by co-run peers program instrumentation or the analyzer code and usable for interference with any peers (which may be unknown at the time of footprint analysis). With the new theory in this thesis, footprint can be obtained with near real-time efficiency.

In an offline solution, Jiang *et al.*[105] defined the concept of *politeness* for a program as "the reciprocal of the sum of the degradations of all co-run groups that include the job." The politeness is measured by the effect on the execution time, not just the miss ratio, and is used to approximate optimal job scheduling.

In an online solution, the high cost of co-run testing is addressed in a strategy called Bubble-Up[112]. The strategy has two steps. First, a program is co-run against an expanding bubble to produce a sensitivity curve. The bubble is a specially designed probe program. In the second step, the pressure of the program is reported by another probe and probing run. Bubble-Up is a composable strategy, since each program is tested individually without testing all program combinations. Bubble-Up extracts the factors that determine the program execution time. In comparison, the footprint theory has a narrower scope, which includes just the factors that determine the program behavior in cache.

Two recent solutions use machine learning. Delimitrou and Kozyrakis[113] built a data center scheduler called Paragon. The design of Paragon identifies 10 sources of interference. It uses offline training (through probe programs) to build parameterized models on their performance impact. During online use, Paragon feeds the history information to a learning algorithm

⑥Journal version of [93-94].

called collaborative filtering. Collaborative filtering supports sparse learning. Based on a small amount of past data, it can predict application-application interference and application-machine match.

Statistical techniques have had many uses in performance analysis of parallel code, including clustering, factoring, and correlation[114], linear models (with nonlinear components)[115], queuing models[116], directed searches[117], and analytical models[118].

Machine learning is general and can consider different types of resources together. It is also scalable as more factors can be added by having additional learning. Paragon's learning technique observes the co-run results but has to be given the solo-run speed to compute the co-run slowdown. The cache model complements performance models, which can include the specialized model as a component. Locality metrics such as the footprint can be used as an input to a learning algorithm. While the strength of machine learning is the breadth and the general framework, the strength of the locality theory is the depth and the focused formulation. As a benefit of the latter, we now can understand the shared cache with mathematically tractable models and derive precise co-run miss ratios.

3.7.5 Performance Scaling Models

Using the PRD/CRD model[13], Wu *et al.*[14] conducted experiments on a wide range of symmetric multithreaded benchmarks on modest problem size and core counts and used their scaling framework to study the performance (average memory access time AMAT) over cache hierarchy scaling for large problem sizes on large-scale (LCMPs). The study focuses on the effect of hardware characteristics such as core counts, cache sizes, and cache organizations on different programs and program inputs, but not on hardware independent program characterization.

3.8 Related Techniques

3.8.1 Input-Centric Analysis

The early work in profiling examines multiple executions to identify what behavior is common. For example, Wall compared the hot variables and functions found in different executions of the same program[119]. Recent work has gone one step further to identify the patterns of change and predict how the behavior will differ from run to run. Shen called it *input-centric analysis*[120].

Input-centric analysis covers the intermediate ground between dynamic analysis, which is for a single execution, and static analysis, which is for all executions. For problems such as reuse distance and footprint, dynamic analysis is too specific, because the result is limited to what happens in one execution. Static analysis is too general, since it assumes all code paths are possible. Input-centric analysis provides a way to overcome these limitations.

Imperative to input-centric analysis is a metric whose results can be compared between different executions. The access of a memory location, for example, is not comparable because a program may allocate the same datum to different locations in different runs. Neither is the instruction making the access, since the same access may be made from different codes in different runs. Reuse distance is the first metric to enable input-centric analysis, since it is not tied to specific memory allocation or control flow and can be compared between different runs.

The first group of work studied how the reuse distance changes in different runs and developed statistical models of locality prediction (called whole-program locality)[5,36,121], miss-rate prediction[28], performance prediction (not just cross-input but also cross-architecture)[25,122], critical load instruction prediction[29], and locality phases[123-125]. Zhong *et al.* surveyed these and other techniques and categorized them as behavior (rather than code) based analysis, analogous to observation and prediction in the physical and biological sciences[5].

More recent work combined behavior and code analysis, in particular, showed how to predict the loop bounds in different runs. To characterize program inputs, Mao and Shen defined an extensible input characterization language (XICL)[126]. Jiang *et al.* defined the notion of seminal behavior, which is the smallest set of program values that collectively determine the iteration count of all loops[127]. Learning techniques such as classification trees were used to identify the seminal behavior[126,128]. Wu *et al.* later expanded the loop analysis to capture sequence patterns[129].

Input-centric analysis has been used to improve the feedback-driven program optimization (FDO) in the IBM XL C compiler[127] and the just-in-time (JIT) compiler in Java virtual machines[120,130-131]. Profiles from different inputs are routinely used in feedback-driven and iterative compiler optimization. The quality of optimization depends on the quality of profiles. The dependence has been examined using statistics[132-133].

3.8.2 Profiling and Performance Monitoring

The term profiling broadly refers to techniques that extract and analyze a subset of events in a program execution. Locality profiling extracts and analyzes the sequence of memory accesses. It does so by program instrumentation. At each memory reference, it inserts

a call to pass the memory address to an analyzer. The instrumentation can be done at source or binary level. Source level instrumentation is made by a compiler such as GCC, Open64, and LLVM, usually at the level of the intermediate code. Binary instrumentation is by a binary rewriting tool. Both can be done statically, i.e., without running a program. Binary rewriting can also be done dynamically when a program is running.

The main problem of profiling is the cost of the instrumentation. A compiler can optimize the instrumented code statically. Another advantage is that the instrumentation tool is portable if a compiler is portable. In comparison, binary rewriting is architecture specific. For example, ATOM instruments only Alpha binary[134], and Pin x86 binary[135]. On the other hand, Pin can instrument dynamically loaded library, which a static tool cannot do.

Profiling does not model the timing effect, for which we need to either monitor an execution on actual hardware or reproduce it in a simulator.

Performance monitoring for parallel code has a long history[136-138]. Modern processors provide hardware counters to monitor hardware events with little or no run-time cost. The events related to memory performance include the frequency of cache misses, cache coherence misses, and various cycle counts, including stalled cycles. When many events are being monitored in a large system over a long execution, the large volume of results presents two problems. The first is the time and space cost of collecting and storing these results. The second is analysis — how to identify high-level information from low-level measurements.

These problems are solved by monitoring and visualization tools, including commercial ones such as Intel VTune Amplifier, AMD CodeAnalysist, and CrayPat, and open-source projects such as PAPI library[139], HPCToolkit[140], TAU[141], and Open|SpeedShop[142]. The aggregation of information is usually code centric, which shows performance in program functions and instructions. Vertical profiling identifies performance problems across a software stack[143]. Continuous program optimization (CPO) not only finds performance problems but also optimizes performance automatically[58,60,144-145]. In recent work, data-centric aggregation is used to pin-point locality problems more effectively, for issues of not just cache misses but also non-uniform memory access (NUMA) latency[146-148].

Bursty Sampling and Shadow Profiling. Arnold and Ryder pioneered a general framework to sample Java code, i.e., the first few invocations of a function or the beginning iterations of a loop[56]. It has been adopted for hot-stream prefetching in C/C++ in bursty sampling [20] and extended to sample both static and dynamic bursts for calling context profiling[149]. Shadow profiling pauses a program at preset intervals and forks a separate process to profile in parallel with the base program[64-65]. The reuse distance analysis is not a good target for these techniques because of the uncertain length of the reuse windows. However, the footprint can be easily sampled using shadow profiling. Reuse distance can then be computed using the conversion theory.

4 Conclusions

In this paper we have described the recent footprint theory of locality, including the definition and formal properties especially the footprint composition and the conversion between window-based statistics, i.e., the footprint, and reuse-based statistics, e.g., the miss ratio. We have surveyed a large literature, more than 140 publications over the past four decades, focusing on the working set theory, which lays the foundation of this research field, and recent performance models, which address the complex challenges posed by the modern multicore memory hierarchy. Through the review, we have appraised their strengths and weaknesses and pointed out the relation with the new footprint theory.

Nicholas Wirth titled his 1976 book "Algorithms + Data Structures = Programs" to emphasize the core subjects and their relations. We would modernize the figurative equation for use on today's machines and say "(Algorithms + Data Structures) × Locality = Efficient Programs". In theory, locality is fundamental in understanding the nature of computation. In practice, memory optimization is necessary in the design and use of every computing system. Locality research has made tremendous progress and immense impacts. This review focuses on the growing body of research to uncover the essential aspects of program behavior in shared cache and as a result enhance our ability to understand and manage program interaction on multicore systems.

Acknowledgment We thank Peter Denning and Xipeng Shen for always patiently and promptly answering our questions about their work, for the many people who worked with us at Rochester, and for the reviewers and the organizers of this special issue. Given the scope and depth of the past research, it is inevitable that the presentation fails to be complete and completely precise. We apologize for any omission and misrepresentation. Any error is entirely ours. We appreciate reader feedback, which can be sent to the email address listed in the first page of the paper.

A Chinese version of the first two sections have been co-authored with Yuan Liang and published in the Journal of Computer Engineering and Science[150].

References

[1] Zhang X, Dwarkadas S, Shen K. Towards practical page coloring-based multicore cache management. In *Proc. the EuroSys Conference*, April 2009, pp.89-102.

[2] Denning P J. Working sets past and present. *IEEE Transactions on Software Engineering*, 1980, 6(1): 64-84.

[3] Denning P J. The working set model for program behaviour. *Communications of the ACM*, 1968, 11(5): 323-333.

[4] Brock J, Luo H, Ding C. Locality analysis: A nonillion time window problem. In *Proc. Big Data Analytics Workshop*, June 2013.

[5] Zhong Y, Shen X, Ding C. Program locality analysis using reuse distance. *ACM TOPLAS*, 2009, 31(6): 1-39.

[6] Zhong Y, Orlovich M, Shen X, Ding C. Array regrouping and structure splitting using whole-program reference affinity. In *Proc. PLDI*, June 2004, pp.255-266.

[7] Ding C, Chilimbi T. All-window profiling of concurrent executions. In *Proc. the 13th PPoPP* (Poster Paper), Feb. 2008, pp.265-266.

[8] Xiang X, Bao B, Bai T, Ding C, Chilimbi T M. All-window profiling and composable models of cache sharing. In *Proc. PPoPP*, Feb. 2011, pp.91-102.

[9] Xiang X, Bao B, Ding C, Gao Y. Linear-time modeling of program working set in shared cache. In *Proc. PACT*, Oct. 2011, pp.350-360.

[10] Xiang X, Ding C, Luo H, Bao B. HOTL: A higher order theory of locality. In *Proc. ASPLOS*, March 2013, pp.343-356.

[11] Xiang X, Bao B, Ding C, Shen K. Cache conscious task regrouping on multicore processors. In *Proc. the 12th CCGrid*, May 2012, pp.603-611.

[12] Xiang X. A higher order theory of locality and its application in multicore cache management [Ph.D. Thesis]. Computer Science Dept., Univ. of Rochester, 2014.

[13] Wu M, Yeung D. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proc. PACT*, Oct. 2011, pp.264-275.

[14] Wu M, Zhao M, Yeung D. Studying multicore processor scaling via reuse distance analysis. In *Proc. the 40th ISCA*, June 2013, pp.499-510.

[15] Thiébaut D, Stone H S. Footprints in the cache. *ACM Transactions on Computer Systems*, 1987, 5(4): 305-329.

[16] Suh G E, Devadas S, Rudolph L. Analytical cache models with applications to cache partitioning. In *Proc. the 15th ICS*, June 2001, pp.1-12.

[17] Chandra D, Guo F, Kim S, Solihin Y. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. the 11th HPCA*, Feb. 2005, pp.340-351.

[18] Belady L A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 1966, 5(2): 78-101.

[19] Denning P J. Thrashing: Its causes and prevention. In *Proc. AFIPS Fall Joint Computer Conference, Part* 1, Dec. 1968, pp.915-922.

[20] Chilimbi T M, Hirzel M. Dynamic hot data stream prefetching for general-purpose programs. In *Proc. PLDI*, June 2002, pp.199-209.

[21] Mattson R L, Gecsei J, Slutz D, Traiger I L. Evaluation techniques for storage hierarchies. *IBM System Journal*, 1970, 9(2): 78-117.

[22] Jiang S, Zhang X. LIRS: An efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proc. SIGMETRICS*, June 2002, pp.31-42.

[23] Smith A J. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proc. the 2nd ICSE*, Oct. 1976, pp.286-292.

[24] Hill M D, Smith A J. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 1989, 38(12): 1612-1630.

[25] Marin G, Mellor-Crummey J. Cross architecture performance predictions for scientific applications using parameterized models. In *Proc. SIGMETRICS*, June 2004, pp.2-13.

[26] Snir M, Yu J. On the theory of spatial and temporal locality. Technical Report, DCS-R-2005-2564, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 2005.

[27] Fang C, Carr S, Önder S, Wang Z. Path-based reuse distance analysis. In *Proc. the 15th CC*, Mar. 2006, pp.32-46.

[28] Zhong Y, Dropsho S G, Shen X, Studer A, Ding C. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, 2007, 56(3): 328-343.

[29] Fang C, Carr S, Önder S, Wang Z. Instruction based memory distance analysis and its application to optimization. In *Proc. PACT*, Sept. 2005, pp.27-37.

[30] Beyls K, D'Hollander E H. Discovery of locality-improving refactorings by reuse path analysis. In *Proc. the 2nd Int. Conf. High Performance Computing and Communications*, Sept. 2006, pp.220-229.

[31] Beyls K, D'Hollander E H. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *Proc. the 3rd ACM Conference on Computing Frontiers*, May 2006, pp.373-382.

[32] Kelly T, Cohen I, Goldszmidt M, Keeton K. Inducing models of black-box storage arrays. Technical Report, HPL-2004-108, HP Laboratories Palo Alto, 2004.

[33] Almeida V, Bestavros A, Crovella M, de Oliveira A. Characterizing reference locality in the WWW. In *Proc. the 4th International Conference on Parallel and Distributed Information Systems* (*PDIS*), December 1996, pp.92-103.

[34] Bennett B T, Kruskal V J. LRU stack processing. *IBM Journal of Research and Development*, 1975, 19(4): 353-357.

[35] Olken F. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report, LBL-12370, Lawrence Berkeley Laboratory, 1981.

[36] Ding C, Zhong Y. Predicting whole-program locality through reuse distance analysis. In *Proc. PLDI*, June 2003, pp.245-257.

[37] Zhong Y, Ding C, Kennedy K. Reuse distance analysis for scientific programs. In *Proc. Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, March 2002.

[38] Schuff D L, Kulkarni M, Pai V S. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proc. the 19th PACT*, Sept. 2010, pp.53-64.

[39] Kim Y H, Hill M D, Wood D A. Implementing stack simulation for highly-associative memories. In *Proc. SIGMETRICS*, May 1991, pp.212-213.

[40] Sugumar R A, Abraham S G. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Technical Report, University of Michigan, August 1993.

[41] Burger D, Austin T. The SimpleScalar tool set, version 2.0. Technical Report, CS-TR-97-1342, Department of Computer Science, University of Wisconsin, June 1997.

[42] Almasi G, Cascaval C, Padua D A. Calculating stack distances efficiently. In *Proc. the ACM SIGPLAN Workshop on Memory System Performance*, June 2002, pp.37-43.

[43] Denning P J, Schwartz S C. Properties of the working set model. *Communications of the ACM*, 1972, 15(3): 191-198.

[44] Berg E, Hagersten E. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proc. ISPASS*, March 2004, pp.20-27.

[45] Berg E, Hagersten E. Fast data-locality profiling of native execution. In *Proc. SIGMETRICS*, June 2005, pp.169-180.

[46] Eklov D, Hagersten E. StatStack: Efficient modeling of LRU caches. In *Proc. ISPASS*, March 2010, pp.55-65.

[47] Eklov D, Black-Schaffer D, Hagersten E. Fast modeling of shared caches in multicore systems. In *Proc. the 6th*

HiPEAC, Jan. 2011, pp.147-157.

[48] Shen X, Shaw J, Meeker B, Ding C. Locality approximation using time. In *Proc. the 34th POPL*, Jan. 2007, pp.55-61.

[49] Shen X, Shaw J. Scalable implementation of efficient locality approximation. In *Proc. the 21st LCPC Workshop*, July 31-August 2, 2008, pp.202-216.

[50] Jiang Y, Zhang E Z, Tian K, Shen X. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proc. the 19th CC*, Mar. 2010, pp.264-282.

[51] Shen X, Shaw J, Meeker B, Ding C. Locality approximation using time. Technical Report, TR 901, Department of Computer Science, University of Rochester, December 2006.

[52] Jiang Y, Tian K, Shen X. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proc. HiPEAC*, Jan. 2010, pp.201-215.

[53] West R, Zaroo P, Waldspurger C A, Zhang X. Online cache modeling for commodity multicore processors. *Operating Systems Review*, 2010, 44(4): 19-29.

[54] Fedorova A, Seltzer M, Smith M D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. the 16th PACT*, Sept. 2007, pp.25-38.

[55] Zhou S. An efficient simulation algorithm for cache of random replacement policy. In *Proc. the IFIP Int. Conf. Network and Parallel Computing*, Sept. 2010, pp.144-154.

[56] Arnold M, Ryder B G. A framework for reducing the cost of instrumented code. In *Proc. PLDI*, June 2001, pp.168-179.

[57] Hirzel M, Chilimbi T M. Bursty tracing: A framework for low-overhead temporal profiling. In *Proc. ACM Workshop on Feedback-Directed and Dynamic Optimization*, Dec. 2001.

[58] Cascaval C, Duesterwald E, Sweeney P F, Wisniewski R W. Multiple page size modeling and optimization. In *Proc. the 14th PACT*, Sept. 2005, pp.339-349.

[59] Zhong Y, Chang W. Sampling-based program locality approximation. In *Proc. the 7th ISMM*, June 2008, pp.91-100.

[60] Tam D K, Azimi R, Soares L, Stumm M. RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations. In *Proc. the 14th ASPLOS*, Mar. 2009, pp.121-132.

[61] Niu Q, Dinan J, Lu Q, Sadayappan P. PARDA: A fast parallel reuse distance analysis algorithm. In *Proc. IPDPS*, May 2012.

[62] Cui H, Yi Q, Xue J, Wang L, Yang Y, Feng X. A highly parallel reuse distance analysis algorithm on GPUs. In *Proc. the 26th IPDPS*, May 2012, pp. 1284-1294.

[63] Gupta S, Xiang P, Yang Y, Zhou H. Locality principle revisited: A probability-Based quantitative approach. In *Proc. the 26th IPDPS*, May 2012, pp.995-1009.

[64] Moseley T, Shye A, Reddi V J, Grunwald D, Peri R. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proc. CGO*, March 2007, pp.198-208.

[65] Wallace S, Hazelwood K. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. CGO*, Mar. 2007, pp.209-220.

[66] Cascaval C, Padua D A. Estimating cache misses and locality using stack distances. In *Proc. the 17th ICS*, June 2003, pp.150-159.

[67] Allen R, Kennedy K. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers, 2001.

[68] Beyls K, D'Hollander E H. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 2005, 51(4): 223-250.

[69] Pugh W, Wonnacott D. Eliminating false data dependences using the Omega test. In *Proc. PLDI*, June 1992, pp.140-151.

[70] Chauhan A, Shei C Y. Static reuse distances for locality-based optimizations in MATLAB. In *Proc. the 24th ICS*, June 2010, pp.295-304.

[71] Shen X, Gao Y, Ding C *et al.* Lightweight reference affinity analysis. In *Proc. the 19th ICS*, June 2005, pp.131-140.

[72] Bao B, Ding C. Defensive loop tiling for shared cache. In *Proc. CGO*, Feb. 2013, pp.1-11.

[73] Bao B. Peer-aware program optimization [Ph.D. Thesis]. Computer Science Dept., Univ. of Rochester, January 2013.

[74] Yuan L, Ding C, Štefankovič D, Zhang Y. Modeling the locality in graph traversals. In *Proc. the 41st ICPP*, Sept. 2012, pp.138-147.

[75] Agarwal A, Hennessy J L, Horowitz M. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 1988, 6(4): 393-431.

[76] Ding C, Chilimbi T. A composable model for analyzing locality of multi-threaded programs. Technical Report, MSR-TR-2009-107, Microsoft Research, August 2009.

[77] Strohmaier E, Shan H. APEX-Map: A parameterized scalable memory access probe for high-performance computing systems. *Concurrency and Computation: Practice and Experience*, 2007, 19(17): 2185-2205.

[78] Ibrahim K Z, Strohmaier E. Characterizing the relation between Apex-Map synthetic probes and reuse distance distributions. In *Proc. ICPP*, Sept. 2010, pp.353-362.

[79] He L, Yu Z, Jin H. FractalMRC: Online cache miss rate curve prediction on commodity systems. In *Proc. IPDPS*, May 2012, pp.1341-1351.

[80] Saltzer J H. A simple linear model of demand paging performance. *Communications of the ACM*, 1974, 17(4): 181-186.

[81] Strecker W D. Transient behavior of cache memories. *ACM Transactions on Computer Systems*, 1983, 1(4): 281-293.

[82] King W F. Analysis of demand paging algorithms. In *Proc. IFIP Congress*, August 1971, pp.485-490.

[83] Fagin R, Price T G. Efficient calculation of expected miss ratios in the independent reference model. *SIAM Journal of Computing*, 1978, 7(3): 288-297.

[84] Dan A, Towsley D F. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proc. SIGMETRICS*, May 1990, pp.143-152.

[85] Gu X, Ding C. Reuse distance distribution in random access. Technical Report, URCS #930, University of Rochester, January 2008.

[86] Denning P J, Slutz D R. Generalized working sets for segment reference strings. *Communications of the ACM*, 1978, 21(9): 750-759.

[87] Easton M C, Fagin R. Cold-start vs. warm-start miss ratios. *Communications of the ACM*, 1978, 21(10): 866-872.

[88] Shedler G, Tung C. Locality in page reference strings. *SIAM Journal on Computing*, 1972, 1(3): 218-241.

[89] Stone H S, Turek J, Wolf J L. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 1992, 41(9): 1054-1068.

[90] Thiébaut D, Stone H S, Wolf J L. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 1992, 41(6): 665-676.

[91] Falsafi B, Wood D A. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 1997, 7(1): 104-130.

[92] Wu M J, Yeung D. Identifying optimal multicore cache hierarchies for loop-based parallel programs via reuse distance analysis. In *Proc. the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, June 2012, pp.2-11.

[93] Fedorova A, Blagodurov S, Zhuravlev S. Managing contention for shared resources on multicore processors. *Communications of the ACM*, 2010, 53(2): 49-57.

[94] Zhuravlev S, Blagodurov S, Fedorova A. Addressing shared resource contention in multicore processors via scheduling. In *Proc. ASPLOS*, March 2010, pp.129-142.

[95] Blagodurov S, Zhuravlev S, Fedorova A. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems*, 2010, 28(4): Article No.8.

[96] Chen X E, Aamodt T M. A first-order fine-grained multi-threaded throughput model. In *Proc. HPCA*, Feb. 2009, pp.329-340.

[97] Xie Y, Loh G H. Dynamic classification of program memory behaviors in CMPs. In *Proc. CMP-MSI Workshop*, June 2008.

[98] Hennessy J L, Patterson D A. Computer Architecture: A Quantitative Approach (4th edition). Morgan Kaufmann, 2006.

[99] Sun X H, Wang D. APC: A performance metric of memory systems. *ACM SIGMETRICS Performance Evaluation Review*, 2012, 40(2): 125-130.

[100] Zhao J, Feng X, Cui H *et al.* An empirical model for predicting cross-core performance interference on multicore processors. In *Proc. PACT*, Sept. 2013, pp.201-212.

[101] Wang W, Dey T, Davidson J W *et al.* DraMon: Predicting memory bandwidth usage of multi-threaded programs with high accuracy and low overhead. In *Proc. HPCA*, Feb. 2014.

[102] Kim M, Kumar P, Kim H, Brett B. Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model. In *Proc. IPDPS*, May 2012, pp.1318-1329.

[103] Zhang X, Zhong R, Dwarkadas S, Shen K. A flexible framework for throttling-enabled multicore management (TEMM). In *Proc. ICPP*, Sept. 2012, pp.389-398.

[104] Liu L, Cui Z, Xing M *et al.* A software memory partition approach for eliminating bank-level interference in multicore systems. In *Proc. PACT*, Sept. 2012, pp.367-376.

[105] Jiang Y, Tian K, Shen X, Zhang J, Chen J, Tripathi R. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *IEEE Trans. Parallel and Distributed Systems*, 2011, 22(7): 1192-1205.

[106] Jiang Y, Shen X, Chen J, Tripathi R. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proc. PACT*, Oct. 2008, pp.220-229.

[107] Snavely A, Tullsen D M. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. ASPLOS*, Nov. 2000, pp.234-244.

[108] Shen K. Request behavior variations. In *Proc. ASPLOS*, Mar. 2010, pp.103-116.

[109] Knauerhase R, Brett P, Hohlt B, Li T, Hahn S. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 2008, 38(3): 54-66.

[110] Denning P J. Equipment configuration in balanced computer systems. *IEEE Transactions on Computers*, 1969, C-18(11): 1008-1012.

[111] Wulf W A. Performance monitors for multi-programming systems. In *Proc. the ACM Symposium on Operating System Principles*, Oct. 1969, pp.175-181.

[112] Mars J, Tang L, Skadron K, Soffa M L, Hundt R. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro*, 2012, 32(3): 88-99.

[113] Delimitrou C, Kozyrakis C. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. ASPLOS*, March 2013, pp.77-88.

[114] Ahn D H, Vetter J S. Scalable analysis techniques for microprocessor performance counter metrics. In *Proc. ACM/IEEE Conf. Supercomputing*, Nov. 2002.

[115] Rodríguez G, Badia R M, Labarta J. Generation of simple analytical models for message passing applications. In *Proc. Euro-Par.*, Aug. 31-Sept. 3, 2004, pp.183-188.

[116] Jacquet A, Janot V, Leung C *et al.* An executable analytical performance evaluation approach for early performance prediction. In *Proc. IPDPS*, April 2003.

[117] Miller B P, Callaghan M D, Cargille J M *et al.* The Paradyn parallel performance measurement tool. *IEEE Computer*, 1995, 28(11): 37-46.

[118] Kerbyson D J, Hoisie A, Wasserman H J. Modelling the performance of large-scale systems. *IEE Proceedings - Software*, 2003, 150(4): 214-222.

[119] Wall D W. Predicting program behavior using real or estimated profiles. In *Proc. PLDI*, June 1991, pp.59-70.

[120] Tian K, Jiang Y, Zhang E Z, Shen X. An input-centric paradigm for program dynamic optimizations. In *Proc. OOPSLA*, Oct. 2010, pp.125-139.

[121] Shen X, Zhong Y, Ding C. Regression-based multi-model prediction of data reuse signature. In *Proc. the 4th Annual Symposium of the Los Alamos Computer Science Institute*, Oct. 2003.

[122] Marin G, Mellor-Crummey J. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proc. the Symposium of the Los Alamos Computer Science Institute*, Oct. 2005.

[123] Shen X, Ding C. Parallelization of utility programs based on behavior phase analysis. In *Proc. the International Workshop on Languages and Compilers for Parallel Computing*, Oct. 2005, pp.425-432.

[124] Shen X, Zhong Y, Ding C. Locality phase prediction. In *Proc. ASPLOS*, Oct. 2004, pp.165-176.

[125] Shen X, Zhong Y, Ding C. Predicting locality phases for dynamic memory optimization. *Journal of Parallel and Distributed Computing*, 2007, 67(7): 783-796.

[126] Mao F, Shen X. Cross-input learning and discriminative prediction in evolvable virtual machines. In *Proc. CGO*, Mar. 2009, pp.92-101.

[127] Jiang Y, Zhang E Z, Tian K *et al.* Exploiting statistical correlations for proactive prediction of program behaviors. In *Proc. the 8th CGO*, April 2010, pp.248-256.

[128] Cavazos J, Moss J E B. Inducing heuristics to decide whether to schedule. In *Proc. PLDI*, June 2004, pp.183-194.

[129] Wu B, Zhao Z, Shen X, Jiang Y, Gao Y, Silvera R. Exploiting inter-sequence correlations for program behavior prediction. In *Proc. OOPSLA*, Oct. 2012, pp.851-866.

[130] Arnold M, Welc A, Rajan V T. Improving virtual machine performance using a cross-run profile repository. In *Proc. OOPSLA*, Oct. 2005, pp.297-311.

[131] Tian K, Zhang E Z, Shen X. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *Proc. OOPSLA*, Oct. 2011, pp.445-462.

[132] Chen Y, Huang Y, Eeckhout L *et al.* Evaluating iterative optimization across 1000 datasets. In *Proc. PLDI*, June 2010, pp.448-459.

[133] Wu B, Zhou M, Shen X *et al.* Simple profile rectifications go a long way — Statistically exploring and alleviating the effects of sampling errors for program optimizations. In *Proc. the European Conference on Object-Oriented Programming*, July 2013, pp.654-678.

[134] Srivastava A, Eustace A. ATOM: A system for building customized program analysis tools. In *Proc. PLDI*, June 1994, pp.196-205.

[135] Luk C, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi V J, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, June 2005, pp.190-200.

[136] Wagner Meira Jr., LeBlanc T, Poulos A. Waiting time analysis and performance visualization in Carnival. In *Proc. ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.

[137] Reed D A, Elford C L, Madhyastha T M, Smirni E, Lamm S E. The next frontier: Interactive and closed loop performance steering. In *Proc. ICPP Workshop*, Aug. 1996, pp.20-31.

[138] Darema-Rogers F, Pfister G F, So K. Memory access patterns of parallel scientific programs. In *Proc. SIGMETRICS*, May 1987, pp.46-58.

[139] Browne S, Dongarra J, Garner N, Ho G, Mucci P. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 2000, 14(3): 189-204.

[140] Adhianto L, Banerjee S, Fagan M, Krentel M, Marin G, Mellor-Crummey J, Tallent N R. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 2010, 22(6): 685-701.

[141] Shende S, Malony A D. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 2006, 20(2): 287-311.

[142] Schulz M, Galarowicz J, Maghrak D, Hachfeld W, Montoya D, Cranford S. Open|SpeedShop: An open source infrastructure for parallel performance analysis. *Scientific Programming*, 2008, 16(2/3): 105-121.

[143] Hauswirth M, Sweeney P F, Diwan A. Temporal vertical profiling. *Software: Practice and Experience*, 2010, 40(8): 627-654.

[144] Childers B, Davidson J, Soffa M L. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *Proc. Symp. Parallel and Distributed Processing*, April 2003.

[145] Cascaval C, Duesterwald E, Sweeney P F, Wisniewski R W. Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development*, 2006, 50(2/3): 239-248.

[146] McCurdy C, Vetter J S. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *Proc. ISPASS*, March 2010, pp.87-96.

[147] Liu X, Mellor-Crummey J M. Pinpointing data locality problems using data-centric analysis. In *Proc. the 9th CGO*, April 2011, pp.171-180.

[148] Liu X, Mellor-Crummey J. A tool to analyze the performance of multithreaded programs on NUMA architectures. In *Proc. the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2014, pp.259-272.

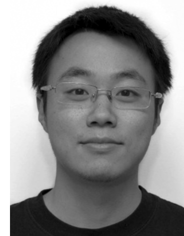
[149] Zhuang X, Serrano M J, Cain H W, Choi J. Accurate, efficient, and adaptive calling context profiling. In *Proc. PLDI*, June 2006, pp.263-271.

[150] Ding C, Yuan L. Program interaction on multicore: Theory and applications. *Computer Engineering and Science*, 2014, 36(1): 1-5. (In Chinese)

Chen Ding received his Ph.D. degree from Rice University, M.S. degree from Michigan Technological University, and B.S. degree from Beijing University, all in computer science before joining University of Rochester in 2000. His research received young investigator awards from NSF and DOE. He co-founded the ACM SIGPLAN Workshop on Memory System Performance and Correctness (MSPC) and was a visiting researcher at Microsoft Research and a visiting associate professor at MIT. He is an external faculty fellow at IBM Center for Advanced Studies.

Xiaoya Xiang graduated in 2005 from Huazhong University of Science and Technology with a B.S. degree in computer science and technology and at the same time from Wuhan University with a B.S. degree in finance. She got her M.S. degree in computer science from Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2008. She earned her Ph.D. degree in computer science at the University of Rochester in 2013. She is now a software engineer at Twitter Inc., where her main focus is the runtime performance of the Twitter services in a cloud environment.

Bin Bao is a senior software engineer at Qualcomm Technologies, Inc. Prior to joining Qualcomm in 2013, Bin spent one year at Adobe Inc. as a computer scientist. He received his Ph.D. degree in computer science from University of Rochester in 2013, M.S. degree in computer science from Institute of Computing Technology, Chinese Academy of Sciences in 2007, and B.S. degree in software engineering from the University of Science and Technology of China in 2004. His current research interests include program analysis and compilation for graphics processors.



Hao Luo is a third year Ph.D. student in the Department of Computer Science, University of Rochester. His research interest lies on performance modeling of multithreaded applications, locality-aware task management, and program behavior analysis.

Ying-Wei Luo received his Ph.D. degree in computer science from Peking University in 1999. He is a full professor of computer science in the School of Electronics Engineering and Computer Science (EECS) in Peking University. His research interests include operating system, system virtualization, and cloud computing.

Xiao-Lin Wang received his Ph.D. degree in computer science from Peking University in 2001. He is now an associate professor of computer science in the School of EECS in Peking University. His research interests include operation system, system virtualization, and cloud computing.

Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT

J. Parallel Distrib. Comput. 64 (2004) 108–134

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Improving effective bandwidth through compiler enhancement of global cache reuse ☆

Chen Ding[a,*] and Ken Kennedy[b]

[a] *Department of Computer Science, University of Rochester, P.O. Box 270226, Rochester, NY 14627, USA*
[b] *Center for High Performance Software Research (HiPerSoft), Rice University, Houston, TX, USA*

Received 20 November 2002; revised 11 September 2003

Abstract

The performance of modern machines is increasingly limited by insufficient memory bandwidth. One way to alleviate this bandwidth limitation for a given program is to minimize the aggregate data volume the program transfers from memory. In this article we present compiler strategies for accomplishing this minimization. Following a discussion of the underlying causes of bandwidth limitations, we present a two-step strategy to exploit global cache reuse—the temporal reuse across the whole program and the spatial reuse across the entire data set used in that program. In the first step, we fuse computation on the same data using a technique called *reuse-based loop fusion* to integrate loops with different control structures. We prove that optimal fusion for bandwidth is NP-hard and we explore the limitations of computation fusion using perfect program information. In the second step, we group data used by the same computation through the technique of *affinity-based data regrouping*, which intermixes the storage assignments of program data elements at different granularities. We show that the method is compile-time optimal and can be used on array and structure data. We prove that two extensions—partial and dynamic data regrouping—are NP-hard problems. Finally, we describe our compiler implementation and experiments demonstrating that the new global strategy, on average, reduces memory traffic by over 40% and improves execution speed by over 60% on two high-end workstations.
© 2003 Elsevier Inc. All rights reserved.

Keywords: Reference affinity; Data locality; Program analysis; Loop fusion; Data transformation; Global cache reuse

1. Introduction

Over the past two decades, the computing power of single-chip microprocessors has increased by a factor of over 6400, in sharp contrast with the much slower rate of improvement for off-chip memory bandwidth, which has increased by a factor of no more than 150 over the same period.[1] To bridge the growing gap between CPU and memory, modern systems employ a hierarchy of cache memory. For the purposes of this paper, we define *effective bandwidth* as the bandwidth at which a memory hierarchy can service a program's demand for data. The goal of the work reported here is to improve effective bandwidth by reducing the total volume of memory access in a program. In the remainder of this section, we discuss the memory bandwidth problem and present the basic ideas underlying our approach.

1.1. The problem of limited memory bandwidth

We measure the *balance* between the data demand of a program and the memory supply of a machine as defined by Callahan et al. [13]. The demand of a program or *program balance* is the number of bytes of memory data the program needs to support a single CPU operation on average. The supply of a machine or *machine balance* is the ratio of the maximal number of bytes the machine can transfer on each cycle to the maximal number of operations the machine can perform on each cycle. If the machine balance is significantly lower than the program balance, the CPU will be

☆ Parts of this work have been published in 2001 and 2000 International Parallel and Distributed Processing Symposium (IPDPS'01 and IPDPS'00) and 1999 International Workshop on Languages and Compilers for Parallel Computing.

*Corresponding author.
E-mail addresses: cding@cs.rochester.edu (Chen Ding), ken@rice.edu (Ken Kennedy).

[1] We derived this estimate based on historical data about CPU speed, pin count, and pin-bandwidth increases compiled by Burger et al. [11].

0743-7315/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2003.09.005

partially idle because the memory system cannot deliver data fast enough to keep it busy on the given program.

As a preliminary to our compiler work, we conducted a performance study in which we measured the machine balance of a 195 MHz MIPS R10K processor on SGI Origin2000 and the program balance of six scientific programs, which include four kernels—*convolution*, *matrix multiply*, *FFT*, and *dmxpy* (matrix-vector multiply from Linpack)—and two full applications—*SP*, a fluid dynamics simulation program from NAS, and *Sweep3D*, a particle transport simulator from DoE. The study used machine parameters, micro-benchmarks, and hardware event counters to measure program and machine balance. All programs were compiled with full optimization from the SGI MIPSpro compiler (except for *matrix multiply*, for which we used a lower optimization level *-O2* because its performance was not memory bound after loop blocking). This study was presented in detail in an earlier paper [20], but we review it briefly here.

Table 1 shows the ratio of program balance to machine balance between four levels of memory hierarchy: registers, level-one cache, level-two cache, and main memory. A ratio of x means that the program needs x times the bandwidth at this memory level to achieve full CPU utilization. Each of these numbers, save one, is significantly greater than one, demonstrating that the bandwidth is insufficient at every level. The numbers in the last column are several times larger than the numbers in the other two columns, establishing that bandwidth from main memory is the most limited. To run at the full CPU speed, these programs would require 3.4 to 10.5 times more memory bandwidth than the SGI delivers. This insufficient bandwidth severely limits program performance. Even in the best case (lowest ratio), the programs cannot on average exceed 33% of the peak CPU performance. The problem is worse in large applications: the average CPU utilization can be no more than 16% for *SP* and 10% for *Sweep3D*.

Table 1
Ratios of program balance (bandwidth demand) to machine balance (bandwidth supply) on SGI Origin2000

Applications	Bandwidth demand vs. supply		
	L1-Reg	L2-L1	Mem-L2
convolution	1.6	1.3	6.5
dmxpy	2.1	2.1	10.5
mmjki (-O2)	6.0	2.1	7.4
FFT	2.1	0.8	3.4
SP	2.7	1.6	6.1
Sweep3D	3.8	2.3	9.8

The bandwidth constraint, as described here, is different from the latency constraint. We define *memory latency* as the time needed for a datum to travel from main memory to CPU without any resource contention. Memory latency can be tolerated by fetching data early. However, prefetching cannot alleviate the bandwidth problem because it does not reduce the aggregate volume of data transfer from memory. In fact, it often aggravates the bandwidth problem by generating unnecessary prefetches. Bandwidth is a fundamental constraint on program performance. For example, if a program needs 10 GB of memory transfer on a machine with 1 GB/s memory bandwidth, the execution would take at least 10 s, even when the machine has zero memory latency, infinite CPU speed, and arbitrarily early and accurate data prefetching.

Previous compiler techniques have fallen far short of solving the bandwidth problem. The SGI Origin had one of the highest memory bandwidths for its time—300 MB/s on a 195 MHz processor. The problem is worse on newer systems because CPU speed is increasing much faster than memory bandwidth is improving. The programs we used in this study were compiled with an excellent commercial compiler that implemented extensive loop and scalar optimizations. Data prefetching was performed by both the hardware and the compiler. Yet, most of these programs could not utilize more than a fraction of the CPU capacity. Thus, the bandwidth constraint has become a principle factor limiting the performance of modern processors.

1.2. A two-step solution strategy

We present a software solution strategy that attempts to minimize the total memory demand of a program. We describe the basic idea of this strategy through an example. Fig. 1(a) shows a sequence of seven accesses to three data elements. Assume that the CPU can either access memory directly for data or access cache for a copy. Given a single-element cache with the commonly

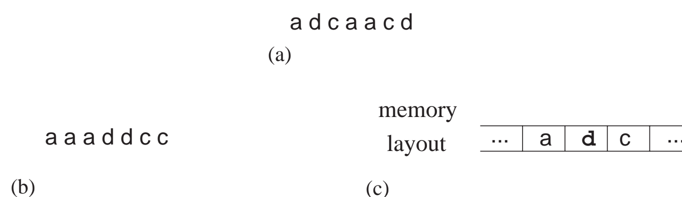


Fig. 1. An example use of our two-step strategy: (a) a sequence of data accesses; (b) fuse computation on the same data; (c) group data used by the same computation.

used LRU replacement policy, we have one cache reuse, which is the third access of element a. The best caching policy stores the data with the closest reuse in the future, as shown by Best in the context of the first Fortran compiler [7] and by Belady for virtual memory [10]. For this example, it would keep a in cache from the beginning and get two cache reuses. This is the best hardware can do, given complete information of data accesses.

By transforming a program in two distinct steps, we can dramatically improve cache performance. The first step is *computation fusion*, which brings together different computations on the same data. For the previous example, it would cluster accesses to the same data element and, assuming no other constraints on access order, produce the sequence in Fig. 1(b). The new sequence has four cache reuses, twice that of the best hardware method.

The second step of our strategy is *data regrouping*, which organizes data elements used by the same computation into contiguous locations in memory. Because cache is organized in non-unit size cache blocks, grouping data in this manner improves bandwidth utilization by ensuring that, when a cache block is loaded, more than one of its data elements is used before it is evicted. In the previous example, the data regrouping step would place the three data elements into contiguous memory locations so that they will occupy the same cache block, as shown in Fig. 1(c).

The two-step transformation achieves optimal caching: useful data are loaded once and only once in a minimal number of cache blocks. Thus the transformed sequence has the lowest possible amount of aggregate memory transfer.

To be effective in large programs, computation fusion must recombine all functions, and data regrouping must re-shuffle the entire data layout. Current compiler techniques are not adequate to carry out these tasks. Most loop transformations target a single loop nest and do not exploit data reuse among disjoint loops. Most data transformations change a single array or object and do not recombine useful data across array and object boundaries.

In what follows, we begin by presenting computation fusion, which attempts to group all accesses to the same datum in a program. We first describe *reuse-based loop fusion*, which fuses loops of different control structures in large applications. Then we prove that optimal fusion for bandwidth is NP-hard. Finally, we examine the limit of computation fusion using perfect program information.

Next, we present data regrouping, which attempts to group all data involved in the same computation. A set of data elements is said to exhibit *reference affinity* if these elements are always used together in a program—that is, if one of the elements is used in a program, the others will be as well. We then present *affinity-based data regrouping*, which ensures that program data elements with reference affinity are allocated together in memory, and we show that the regrouping method presented is compile-time optimal. Then we describe two extensions, partial and dynamic data regrouping, and prove that they are NP-hard problems. We show the use of data regrouping on structure data in addition to array data.

The new global strategies presented here complement rather than replace existing techniques that work on a single loop nest or a single array. Examples of local techniques include unroll-and-jam, loop blocking, loop interchange, loop skewing, and single-array data transformations. Our global techniques target data reuses across loop and array boundaries. They combine loops and arrays but they do not change the relative order of accesses in a loop (except in small ways), nor do they alter the relative placement of data elements in an array. For example, if a loop nest benefits from some local technique, the fused loop nest benefits as well. In fact, the local technique would be more beneficial since the amount of data and data reuse often increase after loop fusion. The global techniques in this article are designed as an independent step before applying local techniques. Joint optimization may produce better results but it is outside the scope of this article.

The next two sections of the paper present computation fusion and data regrouping. The combined strategy is evaluated in Section 4. Section 5 discusses related work and Section 6 provides a summary of the paper's contributions.

2. Reuse-based computation fusion

In this section we cover three main topics: reuse-based loop fusion, a form of computation fusion that focuses on loops; an analysis of the complexity of optimal fusion; and an experimental exploration of the practical limits of fusion given complete program information.

2.1. Reuse-based loop fusion

In most programs, data reuses occur primarily in loops, hence computation fusion equates to loop fusion. This section describes the three components of reuse-based loop fusion: pair-wise fusion, sequential greedy fusion, and multi-level fusion.

2.1.1. Program model

We begin with a program model that abstracts away details that are not relevant to improving data reuse.

- A program consists of a list of loops and non-loop statements. The body of each loop consists of a

similar list. A structured branch is treated as a single meta-statement. A function call is either in-lined or treated as a meta-statement. Programs with unstructured goto statements will not be considered, as these can be eliminated by systematic transformation (see for example Chapter 7 of Allen and Kennedy [5]).

- Each subscript position of an array reference is in one of the two forms: $a[i+t]$ and $a[t]$, where a is the array name, i is a loop index, and t represents a loop-invariant value. Otherwise we assume the subscript ranges over all involved data dimensions.

We use this simple model because it is sufficient for us to test loop fusion on a set of commonly used benchmark programs. It simplifies the description of our fusion algorithms. In principle, we can extend the three new techniques described in this section to optimize more complex loops by using more powerful models such as affine subscript expressions and integer-set mappings, although at the expense of a slower fusion algorithm.

2.1.2. Pair-wise fusion

The loops in real programs often have different control structures, such as single statements, loop nests with different numbers of subloops and perfect or imperfect nesting. Previous fusion techniques have been limited to fusing loops of the same general type and control structure. *Pair-wise fusion* as specified in this section strives to be more general than earlier strategies by fusing two loops at a time based on their data reuse rather than their control structure.

Given two loops, pair-wise fusion analyzes the data access patterns in the outer-most loop and then combines the iterations that share the same data. The two loops can have any control structure. A single statement is considered to be a degenerate loop. Pair-wise fusion classifies data sharing into the following three cases and uses different transformations for each case. Example of pair-wise fusion will be given later in Fig. 2 and Fig. 3.

- *Loop fusion and alignment*, *when data are shared between iterations of the two loops*. It interleaves the data-sharing iterations of the two loops. It may align the loops in two directions. It may shift the second loop down to preserve data dependences, or it may shift the second loop up to bring together data reuse.
- *Loop embedding*, *when data are shared between one loop and one or more iterations of the other loop*. It embeds the former loop into the latter at the earliest data-sharing iteration.
- *Iteration reordering*, *when two loops cannot be fused entirely*. It breaks up the loops and fuses iterations that can be fused. Special cases of iteration reordering include loop splitting and loop reversal. In this paper, we consider only splitting at loop boundaries.

A compiler determines the data sharing between two loops by array section analysis [25]. It analyzes the data access of each iteration of each loop. The iteration access (its data footprint) is parameterized by the loop index of this and all outer loops. It also summarizes the data access patterns of all inner loops. For each dimension of an array, a loop accesses either the whole dimension, a number of elements on the border, or a loop-variant section (a range parameterized by the loop index variable). Data dependence is tested by the intersection of data footprints. The alignment factor is also determined by comparing data footprints.

Algorithm 1 shows the steps of pair-wise fusion. It first determines whether to apply loop fusion, embedding, or splitting. It then finds the alignment factor for fusion and embedding and finally returns the fused loop.

Algorithm 1 Pair-wise fusion

procedure *PairwiseFusion(s, p)*
Require: s and p are either a loop or a non-loop statement
  if s and p do not share data **then**
    return
  end if
  {determine whether to use loop fusion, embedding, or splitting}
  {find the alignment factor for fusion and embedding}
  for each array accessed in both s and p **do**
    find the smallest alignment factor that
      (1) satisfies data dependence, and
      (2) has the closest reuse
    if a constant alignment factor is not possible **then**
      try splitting off boundary iterations
    else
      the alignment factor is infinite
    end if
  end for
  find the largest of all alignment factors, f
  if f is not a constant **then**
    fusion failed
    return
  else
    fuse or embed loops with the alignment f
    return the merged loop and pieces after splitting if any
  end if
end *PairwiseFusion*

When two loops share more than one array, *PairwiseFusion* may have multiple fusion choices. One example is shown in part (a) of Fig. 2. At the top level, we may fuse the two loops at i-level to reuse array A, or we may embed the first loop as one i-iteration to reuse array B. In general, we need to choose between fusion and embedding and between embedding one loop and

```
for i=1, N
  for j=1, N
    D[i,j]=A[i,j]+B[i,j]
  end for
end for

for i=1, N
  for j=1, N
    for k=1, N
      C[i,j]+=A[i,k]+B[k,j]
    end for
  end for
end for
```

(a)

```
for i=2, N
  A[i] -= A[i-1]
end for

A[1] = A[N]

for i=2, N
  A[i] -= A[i-1]
end for
```

(b)

Fig. 2. Two examples of pair-wise fusion: (a) An example showing multiple choices of reused-based fusion; (b) an example showing that fusible relation is not transitive.

embedding the other. We resolve the conflict by choosing the one that reuses the largest arrays or the largest number of arrays. Otherwise, we make a random choice, which would be the case for the example in Fig. 2(a).

One advantage of pair-wise fusion is the efficient test of fusibility. The fusible relation is not transitive. For example for two loops and a statement in Fig. 2(b), any two of the three are fusible but all three together are not, because the alignment required to safely fuse all three would leave no iterations in common between the first and third loop. Since the fusibility of a group of loops cannot be inferred from the fusibility of its subsets, finding all fusion choices would incur a cost exponential to program size. Pair-wise fusion avoids this exponential cost by incremental fusion. However, because it does not examine all possible choices, pair-wise fusion does not always produce the best result. The next section describes a heuristic. Sections 2.2 and 2.3 discuss the problem of optimal fusion and the practical limit of the heuristic-based fusion.

Pair-wise fusion works in the same way for programmer-written loops as for partially fused loops. The two algorithms we present shortly apply pair-wise fusion loop by loop and level by level until no more loops can be fused.

2.1.3. *Single-level sequential greedy fusion*

We use a heuristic we call *sequential greedy fusion*: for every statement or loop from the beginning of a program to the end, we fuse it forward as much as possible toward the previous data-sharing statement or loop. The heuristic is sequential because it considers loops in program order. It is greedy because it tries to merge all the uses of the same data into the place of its first definition. The heuristic is symmetrical to the policy of Best and Belady—while their scheme evicts data that has the furthest reuse, sequential greedy fusion executes the instruction that has the closest reuse. This section applies this heuristic at the source level. Section 2.3 studies its potential by applying it to the execution trace.

Algorithm 2 gives the basic steps of single-level fusion. For each statement s, the algorithm finds the closest predecessor p that shares data with s. Then it invokes pair-wise fusion described in the previous section. If fusion succeeds, the process is repeated for the fused loop because it now accesses a larger set of data and may share data with its predecessors.

Algorithm 2 Single-level sequential greedy fusion

procedure *SingleLevelFusion(p)*
Require: Program p is a list of statements and loops
  for each statement or loop *s[i]* **do**
    GreedyFusion(s[i])
  end for
end *SingleLevelFusion*

procedure *GreedyFusion(s)*
Require: s is a loop or a statement
  search backward from s to find the last data-sharing predecessor p
  if p does not exist or *(s,p)* has been marked as not fusible **then**
    return
  end if
  PairwiseFusion(s, p)
  if fusion failed **then**
    mark *(s,p)* as not fusible, **return**
  else
    {let q be the fused loop}
    GreedyFusion(q)
    for each remaining piece t after splitting **do**
      GreedyFusion(t)
    end for
  end if
end *GreedyFusion*

The example in Fig. 3(a) illustrates the sequential greedy heuristic and pair-wise fusion. The program has two loops sharing access to array A. The loops cannot be fused directly because two intervening statements also access parts of A. *GreedyFusion* examines each loop and statement in program order. It embeds the two statements into the first loop. The two remaining loops are not fusible because $A[1]$ is assigned by the last iteration of the first loop but used by the first iteration of the second loop. *PairwiseFusion* uses iteration splitting to peel off the first iteration of the second loop so that all later iterations can be fused with the first loop. Finally, *PairwiseFusion* uses loop alignment to shift up the iterations of the second loop so that they directly reuse $A[i-1]$. The fused program is shown in Fig. 3(b). The cache locality is greatly improved. Before fusion, most elements of array A cannot be cached if the size of the array is larger than cache. After fusion, most of array A is cached regardless of the size of the array.

Reuse-based loop fusion may add significant instruction overhead because of the inserted branches. Branch statements can be avoided using a code-generation scheme from Allen and Kennedy (Section 8.6.3 of [5]), but at the cost of replicated loop bodies. Future processors will be better equipped to handle branches with features such as predicated execution. Even on current machines, our study in Section 1 shows that programs spend most time waiting for memory, so higher instruction overhead may be tolerated. Section 4 will evaluate reuse-based loop fusion on current machines.

For single-level loops, the algorithm ensures bounded reuse distance for most data reuses in the fused loop, which can then be cached by a constant-size cache regardless of the volume of the input data. We now establish this bound. Our program model allows two forms of array references, $A(c)$ and $A(i+c)$, where i is the loop index and c is a loop invariant. When the number of loop iterations is sufficiently large, most data elements are accessed by array references in the form of $A(i+c)$. Let $A(i_1+c_1)$ and $A(i_2+c_2)$ be two array references in two initial loops with index variables i_1 and i_2, respectively. If two loops are fused with no alignment, the reuses of an A array element are separated by $|c_1-c_2|$ iterations. Now we consider the effect of the alignment. At each step of pair-wise fusion, the alignment factor must be a constant. Suppose that N_{loops} are fused, the alignment is then $O(N_{\text{loops}})$. The reuses of an A element are separated by $O(N_{\text{loops}}+|c_1-c_2|)$ or $O(N_{\text{loops}})$ iterations in the fused loop.

We assume that each initial loop contributes a constant number of references of the form $B(i+c)$ for each array B. Hence, the fused loop has at most $O(N_{\text{loops}})$ references of that form for each array, covering a section of $O(N_{\text{loops}})$ elements in each iteration. The size of the section would be $O(N_{\text{loops}}+k)$ for k consecutive iterations. Replacing k with N_{loops}, we see that the reuses of an A element are separated by at most $O(N_{\text{loops}}+N_{\text{loops}})$ or $O(N_{\text{loops}})$ elements from each array. The maximal reuse distance is bounded by the amount of accessed data from all arrays, which is $O(N_{\text{arrays}}N_{\text{loops}})$.

The upper bound is tight because a worst-case example can be constructed as follows: the body of the first loop is $B_1(i)=A(i-n)$, next are n loops with a body $B_1(i)=B_1(i+1)+B_2(i+1)+\cdots+B_m(i+1)$, finally is a loop with the body $A(i)=B_1(i)$. Let all loops run from 1 to N except for the first loop, which runs from $n+1$ to N. According to our informal proof, the reuse distance between the reuses of an A element will be bounded by nm in the fused loop. This asymptotic bound is the lowest possible because the value of each A array element has to flow through n elements of B_1 array and combine the value of $N-1$ elements of arrays B_2 to B_m before returning to itself. Therefore, the fusion algorithm achieves the tightest asymptotic upper bound on the length of reuse distances in a fused loop.

Being a heuristic, sequential-greedy fusion has two major weaknesses. First, it is not an optimal solution. It fuses loops upward as much as possible and often results in very large loops at the beginning. The second and related problem is that the heuristic is unconstrained, so a fused loop may access too much data and consequently overflow the limited register and cache resources. We have recently developed a method for resource-constrained fusion [29], which could be used to ameliorate this problem, but it is still under evaluation and not used in the results reported here. The evaluation section will measure the effect of the current, unconstrained fusion method.

```
for i=2, N
  A[i] = f(A[i-1])
end for

A[1] = A[N]
A[2] = 0.0

for i=3, N
  B[i] = g(A[i-2])
end for
```

(a)

```
for i=2, N
  A[i] = f(A[i-1])
  if (i==3)
    A[2] = 0.0
  else if (i==N)
    A[1] = A[N]
  end if

  if (i>2 and i<N)
    B[i+1] = g(A[i-1])
  end if
end for
B[3] = g(A[1])
```

(b)

Fig. 3. Examples of pair-wise and sequential greedy fusion: (a) An example program; (b) transformed program.

2.1.4. *Multi-level fusion*

Multi-level fusion first decides the order of loop levels and then applies single-level fusion level by level. It tries

to minimize the total distance of data reuses by minimizing the number of fused loops at outer levels.

Like pair-wise fusion, multi-level fusion is based on the data access patterns rather the control structures of loops. While all data are considered for the correctness of fusion, only large arrays are used to determine the profitability. Different choices lead to different loop fusions. Our current heuristic chooses arrays that have the largest size. It requires that if a loop accesses more than one of the chosen arrays, it must traverse the same data dimension of each of these arrays. Otherwise, it chooses the largest array subset that meets the requirement. In the worst case, it chooses only one array. As a result, each loop level accesses either no data dimension or a unique data dimension of the chosen array(s). Two loop levels may access the same data dimension. In the following description, *data dimension* refers to a dimension of a chosen array.

Algorithm 3 shows the steps of multi-level fusion. For each loop level starting from the outermost, *MultiLevelFusion* determines nesting order at each level L in three steps. First, it examines all data dimensions that are iterated by loops at L or deeper levels. For each data dimension s, it performs a hypothetical analysis in which it first moves all s-traversing loops to level L if possible, then performs sequential greedy fusion, and finally measures the number of fused loops at level L. The analysis in this step is based on the technique developed by McKinley et al., which permutes the loops that access the contiguous data dimension to the innermost level [43]. We use a variation of this method to permute loops that access a chosen data dimension to level L.

Algorithm 3 Multi-level loop fusion

procedure *MultiLevelFusion(S, L)*
Require: S is the set of data dimensions; L is the current loop level
  {Step 1. find the best data dimension for level L}
  for each dimension s in S, test hypothetically **do**
    LoopInterchange(s, L)
    apply *SingleLevelFusion* at level L
    count the number of fused loops
  end for
  chose dimension s' that yields the fewest fused loops
  {Step 2. fuse loops for level L on dimension s'}
  LoopInterchange(s', L)
  apply *SingleLevelFusion* at level L
  {Step 3. continue fusion at level $L+1$}
  for the body of each loop at level L **do**
    MultiLevelFusion(S-s,L+1), s is the data dimension iterated at level L
  end for
end *MultiLevelFusion*

procedure *LoopInterchange(s, L)*
Require: s is a data dimension; L is the current loop level
  for each loop nest **do**
    if loop level t $(t > L)$ iterates data dimension s **then**
      interchange level t to L if possible, otherwise do nothing
    end if
  end for
end *LoopInterchange*

The second step of Algorithm 3 picks the dimension that yields the smallest number of loops after fusion at level L. The third step recursively applies *MultiLevelFusion* at the next loop level. Note that after fusion, not all level-L loops iterate over the same data dimension. Since loop interchange may not always succeed, some level-L loops may access a different data dimension than others. In the algorithm, the dimension s in the third step is not always the dimension s' found in the second step. Loop fusion may take place even if the loops access a dimension other than s'.

We now analyze the time complexity of multi-level loop fusion. Assuming loop splitting at boundary iterations is the only form of iteration reordering, pair-wise fusion takes $O(N_{\text{arrays}})$, and single-level fusion takes $O(N_{\text{init. loops}} * N_{\text{fused loops}} * N_{\text{arrays}})$, where $N_{\text{init. loops}}$ is the number of loops or statements before fusion, $N_{\text{fused loops}}$ is the number of loops after fusion, and N_{arrays} is the number of data arrays in the program. Although this is quadratic in the number of loops in the program, we believe that the running time will be manageable because the number of loops to which fusion will be applied will not usually be large and, the application is developed in a modular style, this number should not grow linearly with the size of the program.

Other types of iteration reordering may increase time complexity of the algorithm. However, boundary splitting is sufficient for the programs used in our evaluation. Since multi-level fusion examines each dimension at each loop level, the total cost is $O(N_{\text{dimensions}})$ times the cost of single-level fusion, where $N_{\text{dimensions}}$ is the number of data dimensions in data arrays.

2.2. Optimal computation fusion

This section formulates the problem of optimal fusion and proves that the problem is NP-hard.

We use the following program model in this section. A program is a sequence of computation units. We assume no cache reuse between different units but perfect cache reuse within the same unit. In other words, when a unit is executed, it loads its data from memory once and only once. Thus, a unit can be thought of as a single loop nest or program section in which all reuse is out of cache.

When two computation units are fused, the fused unit accesses the union of the data of the original units.

At the source level, most programs do not confirm to this model because of branches. However, when a program is executed, it becomes a sequence of operations that can be divided into computation units. Dividing a program into computation units is the job of program analysis. Our goal is to study the fusion rather than the analysis of computation units. We assume that all computation units and their execution order is known. Our result is not limited to loops. A computation unit can be any program code that accesses a large set of data elements, either directly or indirectly through calls to (possibly recursive) functions. Still, we note that most current compilers are best at analyzing large numbers of data accesses only in loop nests.

For our study of computation fusion, we model a program as a hyper graph. The computation units are represented as nodes. Each data element is represented as a hyper edge that connects all nodes in which the element is accessed. In addition, we represent the legality constraint of computation fusion by *bad groups*, which are groups of nodes that cannot be fused together. A set of loops can be fused if they do not include any bad group. Gao et al. [24] and Kennedy and McKinley [31] modeled a program as a normal graph, where each node represents a loop and each edge represents the amount of register reuse between two nodes. Kennedy and McKinley modeled the legality constraint with bad nodes and edges [31]. We use bad groups because the fusible relation is not transitive in pair-wise fusion, as discussed in Section 2.1. We model data dependences as directed edges as in the previous work [24,31].

Definition 1 (Optimal fusion). Given a graph, the fusion minimizes the bandwidth consumption if it fuses the nodes into a sequence of partitions such that

- (Correctness) All dependences between partitions flow forward, and no partition contains a subset that forms a bad group.
- (Optimality) The sum of the node degree of all nodes is minimal, where the degree of a node is the number of hyper edges connected to this node.

Assuming perfect cache reuse within a node and no cache reuse between nodes, the optimality requirement guarantees that the total amount of memory access is minimal over the entire execution. The fused program then consumes a minimal amount of memory bandwidth compared to any version of the program with the same execution time.

The example in Fig. 4 shows why we use a hyper graph instead of a normal graph. The program has six loops. Assuming that all loops can be freely fused except for loops 5 and 6, which form a bad group and cannot be fused together. The bandwidth consumption is measured by the number of arrays read in all loops. The initial program requires a transfer of 20 arrays.

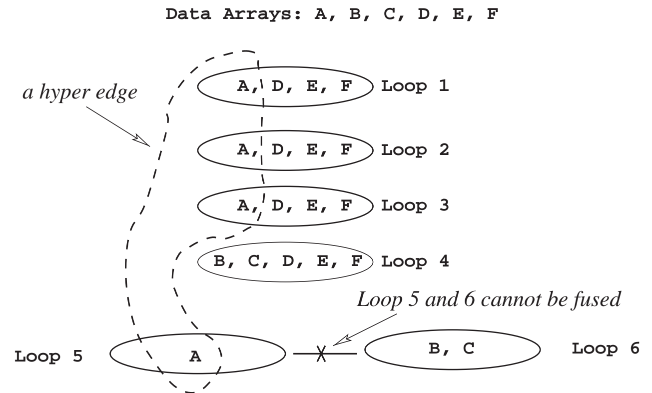


Fig. 4. An example of optimal fusion. The min-cut for a hyper graph fuses loops 1–4 with loop 6, while the min-cut for a normal graph fuses loops 1–4 with loop 5. The former fusion minimizes the bandwidth consumption.

The optimal fusion needs to divide the six nodes into two partitions and minimize the number of hyper edges connecting them. The problem is essentially that of finding a minimal cut on a hyper graph. In this example, the min-cut should put loops 1–4 with loop 6 in one partition and leave loop 5 in the other. The fused program needs a transfer of 7 arrays.

If we were to model register reuse with weighted normal edges and find optimal fusion through a minimal cut on a normal graph [24,31], we would put loops 1–4 with loop 5 in one partition and leave loop 6 in the other. However, this solution does not minimize the bandwidth consumption, because it needs a transfer of 8 arrays instead of 7. This suggests that the problem might be intractable.

The following theorem proves intractability by establishing that general k-way fusion is NP-hard. The proof involves reducing the k-way cut [18] problem to the fusion problem, following the reduction used by Kennedy and McKinley [31]. A direct corollary is that optimal multi-level fusion is also NP-hard. The proof assumes that the number of hyper edges can be the square of the number of nodes, which is not the case in typical programs. If we assume that the number of hyper edges is constant, then k-way cut is polynomial [18].

Theorem 2. *The optimal fusion problem is NP-hard when the number of partitions is greater than two.*

Proof. We reduce k-way cut problem [18] to the fusion problem. Given a graph $G = (V, E)$ and k nodes designated as terminals, k-way cut finds a set of edges of minimal total weight such that removing the edges renders all k terminals disconnected from each other. To convert a k-way cut problem to a fusion problem, we

construct a hyper graph $G' = (V, E)$, where each edge simply becomes a hyper edge connecting exactly two nodes. We assume that each pair of terminals are in a bad group and therefore not fusible. No dependence exists between nodes since there is no directed edge. A k-way cut is minimal in G if and only if the cut gives optimal fusion in G'. Since k-way cut is NP-hard when k is more than two, so is the problem of optimal fusion for bandwidth reduction. □

2.3. Reuse-driven execution

Computation fusion at the source level is often hindered by insufficient information about a program. In this section, we study the limit of computation fusion by assuming it knows the complete execution of a program. We present a technique, *Reuse-driven execution*, that applies sequential greedy fusion to operations in an execution trace. Trace-level reordering gives maximal freedom to computation fusion: it knows precise dependence relation among operations, and it is not restricted by source-level program structures. The results will show the full effect of computation fusion.

Given a program, we first collect its execution trace by instrumenting the source program. A run-time instance of a statement is recorded as an *operation* in the execution trace. We rank program operations based on their issuing cycle on an infinite parallel machine. Then we carry out reuse-driven execution according to Algorithm 4. The effect is to execute the instructions in the trace on a different schedule in which, at each step, priority is given to operations that reuse the data of recently scheduled operations.

The basic scheduling problem addressed by reuse-driven execution is similar to loop fusion. Each trace can be viewed as a fusion graph where nodes represent instructions and hyper edges represent data reuses. We use the algorithm in Fig. 4 because it is efficient enough for us to process large traces. Furthermore, the heuristic is essentially the same as sequential greedy fusion, allowing us to compare the effect of source-level loop fusion with that of trace-level computation fusion later in the evaluation section.

Algorithm 4 Reuse-driven execution

```
procedure ReuseDrivenExecution
  while there exist unexecuted instructions do
    let i be the first such instruction in the ideal parallel
    execution order
    enqueue i to ReuseQueue
    while ReuseQueue is not empty do
      dequeue i from ReuseQueue
      if (i has not been executed) ForceExecute(i)
    end while
  end while
end ReuseDrivenExecution

procedure ForceExecute(j)
  for each operation i that produces operands for j do
    if (i has not been executed) ForceExecute(i)
  end for
  execute j
  for each operand d used by j do
    find the next operation m that uses d
    enqueue m into ReuseQueue
  end for
end ForceExecute
```

To measure how a program reuses its data, we use a concept we call *reuse distance* [21,60], which is the number of distinct data elements accessed between two uses of the same datum. Reuse distance is the same as LRU stack distance, defined by Mattson et al. and measured using a stack algorithm in 1970 [42]. We use a different name because it is shorter, and it is measured much faster using a tree instead of a stack [21]. Reuse distance is an inherent property of a program, and it allows quantitative comparison between programs without being tied to any particular machine. It measures the number of capacity misses: a data access hits in cache if and only if its reuse distance is smaller than the size of cache. We therefore measure the locality of a program execution by the length histogram of all its reuse distances, which gives the number of capacity misses for all cache sizes. A detailed description of reuse distance and its measurement can be found elsewhere [21,60].

Reuse-driven execution changes data reuse behavior, as shown by the histograms in Fig. 5. A histogram consists of a set of points with integer coordinates on the x-axis. A point at a height y means that y thousands of memory references have a reuse distance between $[2^{(x-1)}, 2^x)$. We link discrete points of a histogram into a curve to emphasize peaks, where large portions of memory references reside. Peaks at the far end of the x-axis are memory references that have long reuse distances. A recent study shows that the reuse distance of these references increases at larger program input sizes [21]. Consequently, these references will be cache misses when the input size is sufficiently large. In this study, we measure the number of references with long reuse distances. The goal of reuse-driven execution is to reduce this number.

Fig. 5 shows four graphs. The upper two are a kernel program *ADI*—which has 8 loops in 4 loop nests—and a full application *SP* (Serial version 2.3)—which has over 218 loops in 67 loop nests. Both programs see a significant reduction in the number of long reuse distances, suggesting a significant reduction in cache misses. The reduction is about 30% for *ADI* and 60%

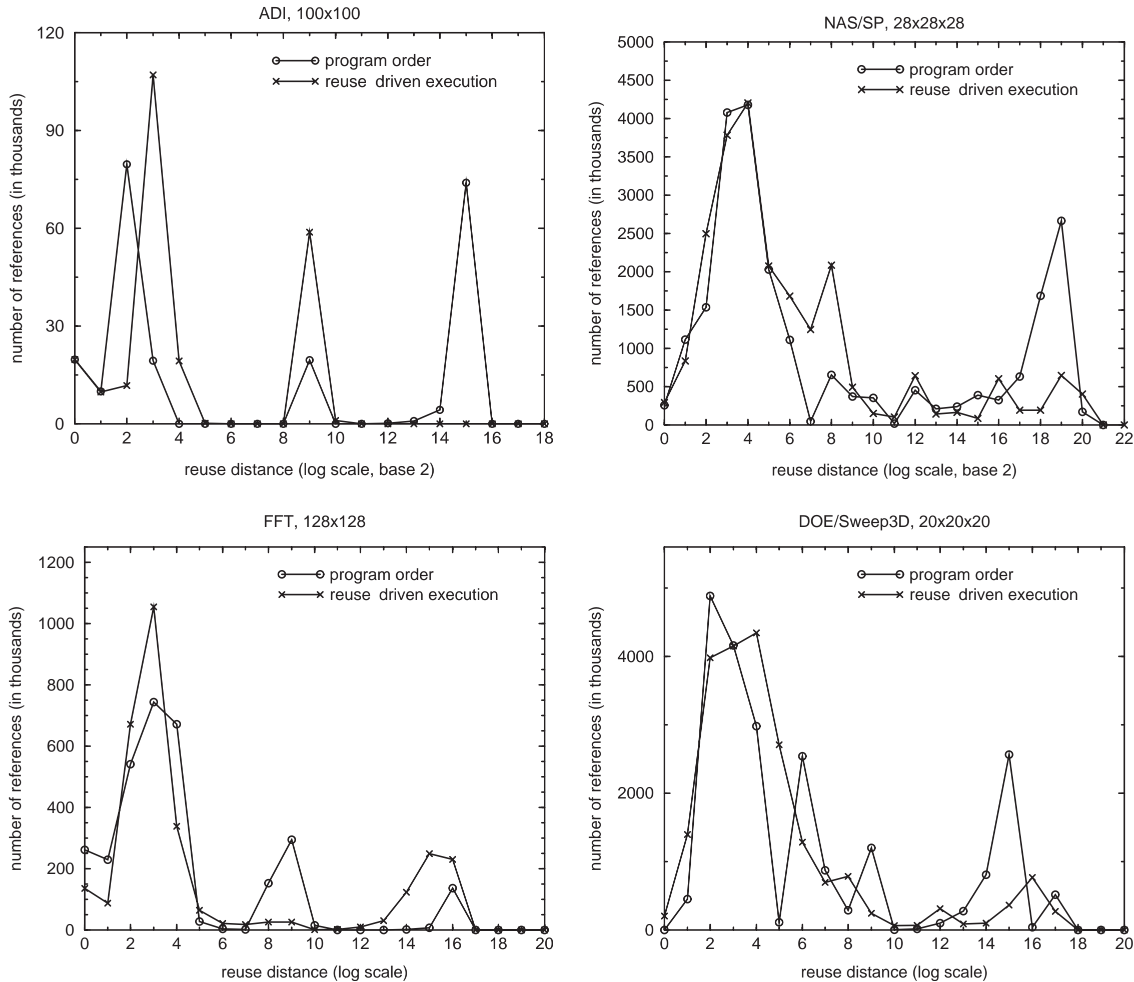


Fig. 5. The effect of reuse-driven execution, shown by the histogram of the length of reuse distances.

for *SP*. The lower two graphs of Fig. 5 show another kernel program, *FFT*, and a full application, *Sweep3D*. Reuse-driven execution did not improve *FFT* (a portion of reuse distances is actually lengthened), but it reduced the number of long reuse distances by 67% in *Sweep3D*. The result of *FFT* shows that as a heuristic, sequential greedy fusion does not always improve locality. We used a few other heuristics, for example, not executing the next reuse if it is too far away in the ideal parallel execution order. But we did not observe any further improvement in locality.

Our experiments with reuse-driven execution demonstrate the potential for improving reuse from cache in applications with a large number of loop nests such as *SP* and *Sweep3D*. Section 4 will compare this result with that of source-level loop fusion.

3. Affinity-based data regrouping

This section presents *affinity-based* data regrouping, which is the second step of our global strategy. While computation fusion improves the temporal locality in data access, data regrouping improves the spatial locality in data storage including cache blocks and memory pages. On today's high-end machines from IBM, SUN, and companies using Intel Itanium and AMD processors, the largest cache in the hierarchy is composed of blocks of no smaller than 64 bytes. If only one four-byte integer is useful in each cache block, 94% of cache space would be occupied by useless data, and only 6% of cache is available for data reuse. A similar issue exists for memory pages, except that the utilization problem can be much worse. Data regrouping reorganizes data based on their reference affinity and packs cache blocks or memory pages with data that are always used together, therefore significantly improving the cache and memory utilization. This section first defines a notion of reference affinity and presents data-regrouping algorithms and their extensions.

3.1. Reference affinity

A set of variables in a program have *reference affinity* if they are always used together in the program. We say

that they are in the same *reference affinity group* or *affinity group* in short. For single data elements, being "used together" means that they are accessed within the same set of consecutive iterations, and the number of the iterations in the set is constant. Reference affinity is reflective, symmetric, and transitive. Therefore, affinity groups form an equivalence partition of program data. We will later extend reference affinity to include sections of data referenced together in outer loops. Large data structures such as arrays may have reference affinity among their elements or sections. The next few subsections will describe single-dimension and multi-dimensional data regrouping, which exploit reference affinity among array elements, array sections, and structure fields. The rest of this subsection use an example to describe the basic idea of data regrouping and its benefits.

The left-hand side of Fig. 6 shows an example program, which traverses a matrix first by rows and then by columns. Assuming we cannot permute loops because of data dependence, one of the loops must access data in large strides, in which case only one element in each cache block is used when the loop bound N is sufficiently large.

The elements of the two arrays have reference affinity because $a[i,j]$ and $b[i,j]$ are always referenced in the same innermost loop. Data regrouping then interleaves them into a single array c, converting $a[i,j]$ into $c[1,i,j]$ and $b[i,j]$ into $c[2,i,j]$ as shown in the right-hand side of Fig. 6. Assuming column-major array layout, the regrouped version guarantees at least two useful numbers in each cache block regardless of the order of data traversal. Therefore, when data access cannot be made fully contiguous by other transformations, data regrouping can further improve cache spatial reuse by combining multiple arrays based on reference affinity.

Data regrouping has three other important effects in addition to improving cache block utilization. These effects are beneficial even in programs where arrays are traversed contiguously. First, regrouping reduces the interference among cache blocks. Data from different arrays—for example, a and b in Fig. 6—may map to the same cache block and cause cache interference misses. Data regrouping will eliminate cache conflicts among members of the same affinity group by placing them in the same cache block.

Second, data regrouping reduces the page-table working set of a program because it combines multiple arrays. It reduces the number of TLB misses when a program accesses more arrays than the available entries in a TLB. On modern machines, TLB misses are time consuming because the CPU halts program execution during a TLB miss. Data regrouping may also improve energy efficiency by reducing the number of active memory pages and lengthening the sleep time for the rest of the memory.

Finally, data regrouping reduces communication cost on shared-memory parallel machines. On these machines, cache blocks are the basis of data coherence and consequently the unit of communication among parallel processors. Good cache-block utilization enabled by data regrouping can better amortize communication latency and utilize communication bandwidth.

3.2. Affinity-based array regrouping

In many applications especially scientific programs, most data are stored in arrays and accessed in loop nests. For these programs, we present a technique called *array regrouping*, which merges arrays based on their reference affinity. We also show that array regrouping can be used to reorganize structure data.

3.2.1. Program analysis

A compiler identifies opportunities of array regrouping in two steps. First, it partitions a program into phases, each of which accesses a data set larger than cache. The idea is to divide a program into

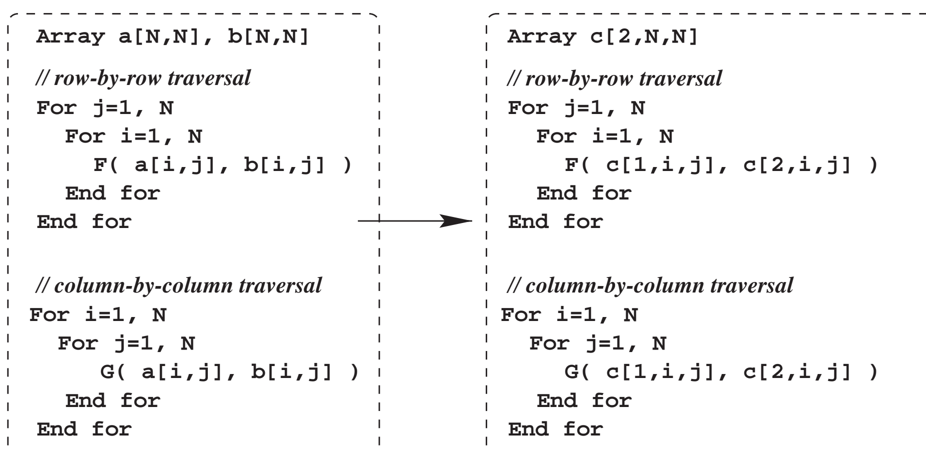


Fig. 6. Example of inter-array data regrouping.

coarse-grained units such that its execution can be viewed as a sequence of coarse-grained steps that do not share data in cache. An innermost loop is a phase if it accesses data larger than cache. An outer loop or a conditional statement is an outer phase if it contains other phases. A branch statement inside an innermost loop is considered as a meta-statement that references the union of the data referenced in all branch paths. The analysis handles function calls in the same way as interprocedural array-section analysis [25]. At the end of the analysis, a program becomes a collection of nested phases.

Since the regrouping algorithm finds groups of data elements that are referenced together in all phases, the execution order and frequency of different phases do not matter. Therefore, the compiler analysis does not need to know the exact control flow among phases. It assumes perfect cache reuse inside a phase and no significant cache reuse between phases. Compiler analysis cannot always accurately determine the amount of data access. However, any error affects only the profitability, not the correctness, of data regrouping. For single-dimension data regrouping, the compiler considers only the innermost phases, which are innermost loops that access a significant amount of data.

The second step of the analysis identifies the sets of compatible arrays. Two arrays are compatible if their sizes differ by at most a constant, and if they are always accessed in the same order in each phase. For example, the size of array $A[N]$ is compatible with $B[N]$ and with $B[N-3]$ but not with $C[N/2]$ or $D[N,N]$. The access order from $A[1]$ to $A[N]$ is compatible with $B[1]$ to $B[N]$ but not with the order from $C[N]$ to $C[1]$ or from $D[1]$ to $D[N/2]$. The second criterion allows compatible arrays to be accessed differently in different phases, as long as they have the same traversal order in the same phase.[2]

The second step uses standard dependence analysis to identify the access order to arrays. For array indirections such as $A[B[i]]$, a compiler can recognize the structure of indirection using a technique described in Ding's dissertation (Section 4.3.2 of [19]). The second step also "unrolls" array dimensions of a constant size. For example, array $A[2,N]$ will be converted into two arrays, $A1[N]$ and $A2[N]$.

Array regrouping is applied to each set of compatible arrays. We assume that grouping incompatible arrays is either impossible or counterproductive. After these preprocessing steps, the question now becomes how to partition compatible arrays into reference affinity groups, given a collection of phases and the set of arrays accessed in each phase.

[2] In general, the traversal order of two arrays needs not to be the same as long as they maintain a consistent relation. For example, array A and B have consistent traversal order if whenever $A[i]$ is accessed, $B[f(i)]$ is accessed, where f is a one-to-one map.

3.2.2. Single-dimension regrouping

Given a set of arrays and a collection of its subsets (phases), two arrays have reference affinity if they are always accessed together, that is, if for any subset containing one array, it must contain the other. The affinity relation is reflexive, symmetric, and transitive; therefore it is a partition. The affinity-based partitioning finds the largest affinity groups because (1) all arrays in each partition have the same reference affinity, and (2) all arrays with the same reference affinity belong to the same partition. In other words, all arrays in a partition belong to the same affinity group, and each affinity group is the largest possible.

The affinity-based partitioning can be determined with efficient algorithms. Algorithm 5 gives one solution. For each array, it encodes its subset membership into a bit vector and then sorts all arrays to find the groups of arrays that are always accessed together. Assuming a total of N_A arrays and N_S phases, the time complexity of this method is $O(N_A * N_S)$.

Algorithm 5 Single-dimension array regrouping

procedure *SingleDimRegrouping*(A, S)
Require: A is the set of all arrays; S is the set of array subsets
  {Step 1. construct a bit vector v for each array}
  for each array a in A and each subset in S **do**
    if (a is in i-th subset of S) $v[i] = 1$ **else** $v[i] = 0$
  end for
  {Step 2. partition arrays}
  sort all bit vectors using radix sort
  group arrays with the same bit vector
end *SingleDimRegrouping*

We discuss single-dimension regrouping through an example—a simplified version of the hydro-dynamics simulation program *Magi* from DOD, which simulates high-impact particle movement in a three-dimensional space. Table 2 lists the six major phases of the program and the attributes accessed in each phase. Attributes of particles are stored in separate arrays and accessed by array indirection. Data access is often not contiguous because particle distribution in space changes during execution. In the worst case, a phase needs to read a cache block for each attribute of each particle. Grouping attribute arrays improves cache-block utilization. For example, if we group *position* and *speed*, the second phase needs to read only one cache block for each particle. However, excessive grouping may hurt cache-block utilization. For example, the same grouping wastes half of each cache block in the first phase because *speed* is not referenced.

Affinity-based data grouping combines arrays in the same affinity group. Algorithm 5 would group attributes

energy, *volume*, and *cumulative totals* in one array, attributes *speed*, *heat*, *derivate*, and *viscosity* in another array, and other attributes each in a different array. Arrays in the same group are always referenced together, so grouping them does not introduce useless data into a cache block. In addition, the size of the reference groups is the largest possible. Any additional regrouping would cause more unused attributes to be accessed by some phase. We will formalize this observation in Section 3.4 and show that data grouping is compile-time optimal.

The partition of reference affinity groups depends on the data references in program phases. When a programmer inserts or deletes data accesses or function calls, the reference affinity may change. Data regrouping needs to be re-applied because the old data layout may degrade performance instead of improving it. This implies that data regrouping should be applied automatically by a compiler, not manually by a programmer; otherwise, the programmer has to change the layout of all program data after any change to any data reference in a program.

Table 2
A simplified view of *Magi*, a hydrodynamics simulation program

	Computation phases	Attributes accessed
1	*building interaction list*	`position`
2	*smoothing attributes*	`position, speed, heat, derivate, viscosity`
3	*hydro-dynamics 1*	`density, momentum`
4	*hydro-dynamics 2*	`momentum, volume, energy, cumulative totals`
5	*stress interaction 1*	`volume, energy, strength, cumulative totals`
6	*stress interaction 2*	`density, strength`

Affinity-based regrouping combines arrays when and only when they are always accessed together. This might seem a bit restrictive in practice. However, many applications use multiple fields of a data structure array together. The algorithm will split each field as a separate array. In addition, aggressive loop fusion often gathers data access of a large number of arrays in a fused loop. Therefore, it should be quite common for two or more arrays to always be accessed together. Later, Section 3.3 discusses methods for relaxing the condition for regrouping at the cost of making the analysis more complex.

3.2.3. Multidimensional regrouping

We now consider programs with nested phases accessing arrays with multiple dimensions. We first motivate multidimensional regrouping with an example in Fig. 7. Part (a) is a loop nest that has one outer loop enclosing two inner loops. Assume that arrays are stored in column-major order. The outer loop traverses the columns of three arrays. The first inner loop references the elements of two arrays A and B together. The second inner loop references only array C. Single-dimension regrouping will recognize the reference affinity between the elements of A and B but cannot exploit the reference affinity between the columns of all three arrays.

Multidimensional regrouping analyzes all levels of loops and finds reference affinity among array sections. In this example, it analyzes the outer loop, puts the columns of three arrays into the same affinity group, and then combines the arrays in two dimensions. Fig. 7(b) shows the transformed program. The new program uses a new array D, whose ith column contains the ith column of all three arrays. Fig. 7(c) shows the new layout for the first column of the three arrays. Arrays A and B are regrouped by elements, and they are

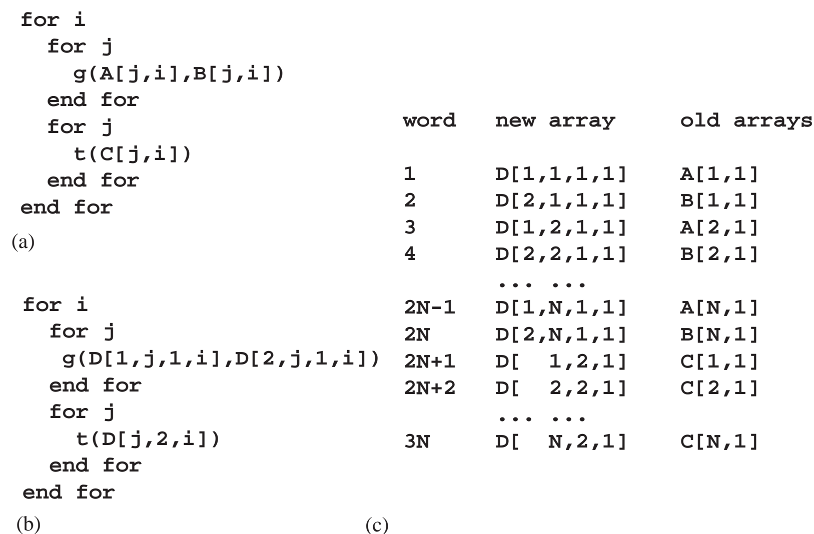


Fig. 7. An example of multidimensional data regrouping.

regrouped with C by columns. The placement order of data elements in the first column of D is the same as the traversal order by the first iteration of the outer loop. After multidimensional regrouping, the program traverses D contiguously, therefore having no cache interference and a single-entry page-table working set.

Programming languages like Fortran do not allow arrays of non-uniform dimensions like those of array D. In addition, the new array indexing created by source-level regrouping may confuse the back-end compiler and negatively affect its register allocation. However, both problems will disappear when regrouping is applied by a back-end compiler, where it should be applied.

Algorithm 6 shows the steps of multidimensional data regrouping. It has two steps. The first collects the set of arrays and their dimensions accessed by each loop. For a loop i and an array a, $Covered(i, a)$ gives the dimension d such that the i accesses ("covers") a on *all* dimensions equal to and lower than d. The analysis traverses loops from the outermost loop and moves inward. For each loop i and array a, it finds $Accessed(i, a, d)$ for all d greater than the dimension of a covered by the outer loop of i, using two criteria. The first is needed in the proof of Theorem 3 for correctness. The second does not affect correctness. It ensures that the algorithm counts only accesses to entire arrays, so that accesses to partial arrays do not distract the regrouping of fully accessed arrays. The resulting *Accessed* set is the set of arrays that accessed together at dimension d by loop i and its outer loops. Other *Accessed* sets find all groups of arrays that are accessed together at some data dimension in all other loops.

The second step of the algorithm applies single-dimension regrouping for each data dimension. For each d, it puts in a set S all $Accessed(i, a, d)$ sets for all i and a. The set is equivalent to the set of phases for dimension d.

Algorithm 6 Multidimensional data regrouping

notation:
A is the set of all arrays, maximal dimension is d_{max}; P is the program; arrays are column-major with dimensions numbered from 1 (right-most); loop levels are numbered from 1 (outermost loop);
Function *Outer* maps from a loop to its closest outer loop; $Outer(i) = P$, if i is an outermost loop
Function *Accessed* maps from triple (loop i, array a, dimension d) to a subset of A, initially maps
  to the empty set
Function *Covered* maps from pair (loop i, array a) to a dimension d,
  $Covered(i, a) =$ the highest d s.t. $Accessed(i, a, d)$ is not empty
  initially $Covered(i, a) = 0$ for all i (including P) and a

procedure *MultiDimRegrouping*
  {Step 1. find arrays and their dimension accessed by each loop}
  for each loop i in a root-first traversal **do**
    for each array a accessed by i **do**
      for each dimension d greater than $Covered(Outer(i), a)$ **do**
        {Step 1.a. find $Accessed(i, a, d)$}
        find $Accessed(i, a, d)$, the set of arrays such that
          1. all arrays are accessed at dimension d and all lower dimensions by loop i and its outer loops, and the same dimension of all arrays is accessed by the same loop
          2. all arrays are accessed at all other dimensions by loops inside i
        {Step 1.b. update $Covered(i, a)$}
        if $Accessed(i, a, d)$ is not empty and $d > Covered(i, a)$ **then**
          $Covered(i, a) = d$
        end if
      end for
    end for
  end for
  {Step 2. partition arrays for each data dimension}
  for each data dimension d between 1 and d_{max} **do**
    let $S = Accessed(i, a, d)$ for all i and a
    $SingleDimRegrouping(A, S)$
  end for
end *MultiDimRegrouping*

We briefly explain the algorithm by the example in Fig. 8, which has two loop nests accessing two arrays. We ignore the constant in subscript expressions, as in single-dimension regrouping. However, it can be easily considered by changing the first condition in Step 1.a to require the same constant in subscript expressions.

Fig. 8 shows the *Accessed* functions found by the algorithm for each data dimension. For the first (most significant) data dimension, the algorithm finds four non-empty *Accessed* sets. If array $a \in Accessed(i, b, d)$ for some i and d, then always $b \in Accessed(i, a, d)$. Hence the four *Accessed* functions represent just two sets. Applying single-dimension data regrouping on these two sets will produce the affinity group shown by the third column. Similarly, the algorithm finds all *Accessed* sets for the other two data dimensions and divides arrays into affinity groups. This example also shows that the algorithm analyzes and groups arrays with a different number of dimensions together.

The correctness of the algorithm is stated in the following theorem, which says that the algorithm fully exploits reference affinity in high dimensional data. A direct corollary is the consistency of the algorithm. Since the algorithm uses single-dimension regrouping for each dimension, it needs to avoid conflicting decisions. In

```
for i
  for j
    for k
      a(j,k,i)
    end for
    b(j,i)
  end for
end for
for i'
  for j'
    for k'
      a(i',j',k')
      b(i',k')
    end for
  end for
end for
```

array dim.	non-empty $Accessed()$ functions	affinity groups
1	$Accessed(i,a,1) = Accessed(i,b,1) = \{a,b\}$ $Accessed(k',a,1) = Accessed(k',b,1) = \{a,b\}$	$\{a,b\}$
2	$Accessed(j,b,2) = \{b\}$ $Accessed(k,a,2) = \{a\}$ $Accessed(k',b,2) = \{b\}$	$\{a\}$ $\{b\}$
3	$Accessed(k,a,3) = \{a\}$ $Accessed(k',a,3) = \{a\}$	$\{a\}$

Other $Accessed()$ functions give empty sets.

Fig. 8. A demonstration of multidimensional regrouping algorithm. The program on the left results in the S sets and $Accessed()$ functions on the right.

particular, the grouping decision at a less significant dimension must not contradict the decision at a more significant dimension. For example, if it decides to group two arrays by elements, then it should not decide, at the column dimension, to separate them.

Theorem 3. *Algorithm* 6 *groups two arrays at dimension d if and only if two arrays are always accessed together at dimensions d and lower. Two arrays, a and b, are accessed together at dimension d and lower, if whenever a is accessed as* $a(i_{d_a},\dots,i_d,\dots,i_2,i_1)$, *b is accessed inside loops* i_1 *to* i_d *as* $b(j_{d_b},\dots,j_{d+1},i_d,\dots,i_2,i_1)$, *where loops* i_1 *to* i_d *may appear in any order or mix with other loops.*

Proof. We first prove the "*if*" part using contradiction. Assume the algorithm groups two arrays, a and b, at dimension d, when at least one loop i accesses the two arrays separately at a dimension d', where $d' \leqslant d$. Assume loop $i_{d'}$ accesses the d' dimension of a but not b. Let i_a be the innermost loop enclosing this reference of a. Examine all loops nested between $i_{d'}$ and i_a. Let loop i' be the innermost loop such that $Covered(i',a) \geqslant d$; therefore, $Covered(Outer(i'),a) \geqslant d - 1$. Loop i' exists because it can at least be i_a and $Covered(i_a,a)$ is equal to the number of dimensions in a. Then Step 1.a finds $Accessed(i',a,d)$ to contain a but not b because the outer loop i does not access the two arrays in the same dimension, a violation of the first condition of Step 1.a. Since $Accessed(i',a,d)$ contains a not b, Step 2 cannot group array a with b at dimension d, a contradiction to the assumption.

The "*only-if*" part says that the algorithm groups two arrays, a and b, at dimension d if they are always accessed together at d and all lower dimensions. We show that if, for some loop i and array c, $Accessed(i,c,d)$ contains one array, say a, it must also contain b. Without the loss of generality, we assume that for some loop i and some array c, Step 1 found a non-empty $Accessed(i,c,d)$ that contains array a. Then loop i and its outer loops access array a at dimension d and all lower dimensions. Considering the third loop of Step 1, we know $d \geqslant Covered(Outer(i),c)$, which means that loop i accesses c at dimension d or lower. Since a belongs to $Accessed(i,c,d)$, loop i must access the same dimension of array a. From the hypothesis, loop i must also access the same dimension of array b. Also from the hypothesis, loop i and its outer loops must access b at d and all lower dimensions in the same way as they do a. Hence $Accessed(i,c,d)$ produced by Step 1 will contain both a and b. Therefore, if some $Accessed(i,c,d)$ contains one array, it must contain the other. Step 2 should always group the two arrays at dimension d. □

Corollary 4. *Algorithm* 6 *produces consistent regrouping, which means that once it determines to group two arrays at dimension d, it must group them at all dimensions lower than d.*

Proof. From Theorem 3, if the algorithm groups two arrays at dimension d, then the program accesses the two arrays together at dimension d and all lower dimensions. This implies that for any dimension d' lower than d, the program accesses the two arrays together at d' and all lower dimensions. Using Theorem 3 again, the algorithm must group the two arrays at dimension d'. Therefore, if the theorem groups two arrays at d, it must group them at any lower dimension d'. □

Multidimensional regrouping fully exploits reference affinity at all granularities. The algorithm subsumes single-dimension regrouping. It extends the definition of compatible arrays, which need not have the same number of dimensions. Only the grouped dimensions need a similar size and access order.

Multidimensional regrouping is especially beneficial in large, complex programs, which often have loops that are not perfectly nested and arrays that are traversed in

different orders. Since data of varied granularity are used together, multidimensional regrouping is necessary to effectively reduce cache interference and the size of page-table working set. Multidimensional regrouping is particularly suitable for programs after aggressive loop fusion, which often produces non-perfectly nested loops that access a large number of data arrays.

3.2.4. Affinity-based structure splitting

Many program have a large number of homogeneous objects, each of which contains the same set of data attributes. In Fortran 77, attributes of an object are stored in different arrays. In languages such as in C, C++, and Java, attributes of an object are stored together in a structure. In neither scheme is the data layout sensitive to the access pattern of a program. The idea that we use to improve the layout of array data can also improve the layout of structure data.

Data regrouping can be used on structure data when the data fields of an object can be split into individual arrays. Examples include particles in a physics simulation and tree nodes in a database. If we know the number of objects, we can store each field in an array. Then we can regroup field arrays based on their reference affinity. The transformation is equivalent to splitting the structure. We call it *affinity-based structure splitting*. In the evaluation section, we will measure the effect of affinity-based structure splitting and compare it with other types of structure layouts.

3.3. Partial and dynamic reference affinity

This section extends data regrouping to consider partial reference affinity—when data are not always used together—and dynamic reference affinity—when different groups of data are used at different times. It also uses data regrouping to minimize memory write-backs.

Partial reference affinity. Reference affinity is partial if the group of references is frequently used together but not always. An example is the first program in Fig. 9. Arrays a and b are used together t times but separately once. The benefit of grouping two arrays will exceed its overhead when t is sufficiently large.

Using partial reference affinity, we gain performance when they are used together but lose cache locality when they are not used together. The overall effect depends on the trade-off between the two. Since both factors are machine dependent, the optimal regrouping is also machine dependent. Assuming we have complete machine information, the problem can be formulated with a weighted, undirected graph, where each data item is a node, and the weight of each edge is the benefit of regrouping minus the overhead. The goal is to pack data that are most beneficial into the same cache block. However, the packing problem is NP-hard because it

```
for step = 1, t
  for i = 1, n
    foo(a[i],b[i])
  end for
end for

for i = 1, n
  bar(a[i])
end for
```

Program I

```
for step = 1, t
  for i = 1, n
    foo(a[i],b[i])
  end for
end for

for step = 1, t
  for i = 1, n
    bar(a[i],c[i])
  end for
end for
```

Program II

Fig. 9. Examples of partial and dynamic reference affinity.

can be reduced from the G-partitioning problem [32], following a similar reduction given by Thabit [54]. Thabit used a weighted, undirected graph to model the frequency when each pair of data are referenced together. He called it a proximity graph [54]. In practice, a compiler can focus on more frequently executed program phases, for example, loops inside a time-step loop.

Dynamic reference affinity. So far, data regrouping produces a single data layout. Alternatively, it can change data regrouping between program phases. In the second program in Fig. 9, for example, the best regrouping is to group a and b at the beginning of the program and then separate these two arrays in the middle. The effect of dynamic regrouping depends on the benefit of data grouping and the cost of run-time regrouping. Both factors are machine dependent, so the optimal data layout is also machine dependent, as in the case of partial reference affinity.

When the machine information is known, the problem is an instance of the problem defined by Kennedy and Kremer [30]. A program is a sequence of phases. Different data layouts result in different execution times for each phase and in different layout-conversion costs between phases. The optimal layout is the one that minimizes the overall execution time. The formulation includes both static and dynamic data regrouping. Kennedy and Kremer proved that the problem is NP-hard and showed that 0-1 integer programming was effective for finding the optimal layout.

Minimizing data write-backs. On machines with insufficient memory bandwidth, data write-backs impede memory read performance because they compete for limited memory bandwidth. Data regrouping can avoid unnecessary write-backs by separating read-only data from modified data. This new requirement can be easily enforced as follows. For each phase, the analysis splits arrays into two disjoint subsets: the first contains read-only data, and the second contains modified data. It treats each as a different phase, and then applies data regrouping. Two arrays will be grouped if and only if they are both read-only or both modified. Data regrouping finds the largest groups that satisfy this

condition. Two arrays being grouped does not mean they must be read-only or modified throughout the whole program. They can be both read-only in some phases and both modified in other phases. When redundant write-backs are allowed, the problem becomes the same as that of partial reference affinity described earlier in this section.

3.4. Optimality

Data regrouping produces largest data groups that are side effect free. It never increases useless data in a cache block anywhere in a program. We can allow side effects by considering partial reference affinity, which increases unused data in cache blocks in some program phase, or by considering dynamic reference affinity, which incurs the cost of reorganization. These side effects vary from machine to machine. If a compiler does not know specific machine parameters or it produces code for different machine configurations, then it should avoid unknown side effects. Our method regroups data if and only if it is always profitable. In this sense, it is compile-time optimal.

Under reasonable assumptions, the optimality can also be defined in terms of the size of the program working set and the amount of program memory transfer. The size of a working set is the total number of memory units occupied by active program data. A memory unit can be a cache block or a virtual memory page. We first discuss the effect of regrouping on cache and then move the discussion to virtual memory. In the case of cache, the working set is minimal if there is no unused data in cache blocks. Data regrouping reduces the size of a working set by clustering useful data into cache blocks. It produces maximal clustering, after which we cannot add any data to any cache block without a side effect.

The size of a working set directly affects memory performance. The more cache blocks a program uses, the more chances of premature eviction of useful data caused by either limited cache capacity or associativity. For convenience, we refer to both cache capacity misses and cache interference misses collectively as *cache overhead misses*. We assume that the number of cache overhead misses is a non-decreasing function of the size of the working set. Intuitively, a smaller working set should never cause more overhead misses than a larger one because the former contains fewer cache blocks. This assumption implies that a minimal working set leads to minimal cache overhead. Because data regrouping minimizes the working set, it minimizes the cache overhead and hence the number of cache capacity and interference misses. Similarly, if virtual memory performance is a non-decreasing function on the page working set, then data regrouping maximizes the performance of virtual memory.

4. Evaluation

4.1. Implementation

We have implemented computation fusion and data regrouping in a version of the D Compiler System from Rice University. The compiler performs whole-program compilation given all source files of an input program. It uses a powerful value-numbering package to handle symbolic variables and expressions inside each subroutine and parameter passing between subroutines. It applies a standard set of analyses, including loop and dependence analysis, data flow analysis, and interprocedural analysis.

A program is processed by four preliminary transformations before loop fusion is applied. The first is procedure in-lining, which brings all computation loops into a single procedure. Procedure in-lining is straightforward for our test programs because their subroutines are not recursive and are called only once in most cases. In other programs, more advanced in-lining will be needed to deal with recursion and code expansion. After procedure in-lining, our compiler splits arrays that have data dimensions of a small constant size. In the process, it needs to unroll the loops that iterate those dimensions. The third step is loop distribution. Finally, the last step propagates constants into loop statements. Our compiler performs loop unrolling, distribution, and constant propagation automatically. In-lining is done by hand; however, it can be automated with additional implementation.

Loop fusion is carried out by applying Algorithm 3. It uses a technique similar to array-section analysis to calculate the data footprint of each loop. Loop interchange is largely unnecessary, as computations are mostly symmetric. One exception is *Tomcatv*, where loop interchange is performed by hand. Iteration reordering is not yet implemented but the compiler signals the places where it is needed. Only one program, *Swim*, requires splitting, which is done by hand.

New code is generated as mappings from the initial iteration space to the fused iteration space. We currently use the *Omega* library [48], which has been integrated into the D compiler [2]. The compilation time is under 1 min for all kernels. For the full application *SP*, however, *Omega* does not scale well with the degree of fusion. It takes 4 min for one-level fusion but 1.5 h for three-level fusion. In contrast, our fusion analysis takes about 2 min for one-level fusion and 4 min for full fusion. If the running time proves excessive, we could generate code using a linear-time algorithm given by Allen and Kennedy (Section 8.6.3 of [5]), but this has not been implemented in our current system.

After fusion, data regrouping is applied level by level on fused loops using Algorithm 6, with two modifications. First, SGI's compiler does a poor job when arrays

are interleaved at the innermost data dimension. So the compiler groups arrays up to the second innermost dimension. This restriction precludes regrouping for reducing memory writebacks. It also results in grouping in less desirable dimensions, as in *Tomcatv*. The other restriction is inherited from the Fortran language, which does not allow non-uniform array dimensions. When multi-level regrouping produced non-uniform arrays, manual changes are made to disable regrouping at outer data dimensions.

4.2. Experimental design

The five test applications are described in Table 3. They come from SPEC, NASA and DOE except for *ADI*, which is a self-written program with separate loops processing boundary conditions to better exercise fusion. Since all programs use iterative algorithms, only the loops inside the time step are timed. The number of cache and TLB misses is measured by hardware counters for the entire execution.

All but one of these programs are measured on an SGI Origin2000 with 250 MHz R12K processors. The exception, *Swim*, is measured on an SGI O2 with a R10K processor in order for a direct comparison with results of another group. Both the R12K and R10K provide hardware counters that measure cache misses with high accuracy. Each has a two-level cache. The level-one (L1) cache uses 32-byte cache lines and is 32 kB in size. The level-two (L2) cache uses 128-byte cache lines and is 1 MB on the O2 and 4 MB on the Origin2000. Both levels of cache are two-way set associative. Both processors hide memory latency through dynamic, out-of-order instruction issuing and compiler-inserted prefetching. The SGI compiler is the MIPSpro Version 7.30. All applications are compiled with the highest optimization flag with prefetching turned on (*f77 -n32 -mips4 -Ofast*), except for *Sweep3D*, on which we use -O2 because it is 2% (23 s) faster than for the original program. The performance improvement at -*Ofast* is similar.

The other SPEC95fp and NAS benchmark programs are omitted because our current implementation cannot process them. The main problem is the lack of long sequences of loop nests due to the extensive use of abstraction by the programmer. To fully expose the computation and data access, we need not only procedure in-lining but an aggressive form of loop unrolling. One such example is *Sweep3d* from DOE. However, for this program we manually unrolled the outermost loops and then fused them at the outermost level by hand.

4.3. Effect of transformations

The effect of optimizations is shown in Fig. 10. The graphs for the first three applications show three sets of bars: the effect of loop fusion alone, the effect of data grouping alone, and the effect of loop fusion plus data regrouping together, all normalized to the original program. For *SP*, one additional set of bars show the effect of fusing one loop level instead of fusing all loop levels. The execution time, the number of graduated instructions, and original miss rates are also given in the figures; however, reductions in cache and TLB are measured on the number of misses, not on the miss rate.

Swim solves a finite-difference model of shallow-water equations. Its core consists of two two-dimensional loops surrounded by statements and one-dimensional loops. The inner loops traverse adjacent columns of 13 arrays. Loop fusion fuses all loop nests with the help of loop splitting and reduces execution time by 10%. The succeeding data grouping merges 13 arrays into 3 and further reduces execution time by 2%, L1 misses by 5%, and TLB by 8%. The performance of *Swim* is reported for SGI O2 because it has the same cache configuration as SGI Octane, a machine used in the work of iteration slicing by Pugh and Rosser [49]. Our fusion achieves the same improvement (10%) as Pugh and Rosser reported for iteration slicing [49]. Our regrouped version is 2% faster than our fused version.

Applying data regrouping only reduces execution time by 16%, even though the data access is mostly contiguous in this program. The improvement comes from a 35% reduction in TLB misses—because regrouping reduces the number of arrays—and a 14% reduction in L2 misses—because grouped arrays have less interference in cache. When applied individually, loop fusion and data regrouping execute 9% and 7% more

Table 3
Description of test programs (*B* denotes a billion and *M* denotes a million)

Name	Source	Input size	Lines/loops/arrays	Inst./loads/stores
Swim	SPEC95	513×513	429/8/15	40.5B/11.0B/4.12B
Tomcatv	SPEC95	513×513	221/18/7	3.33B/720M/289M
ADI	self-written	$2K \times 2K$	108/8/3	599M/155M/92M
SP	NAS/NPB Serial v2.3	class B, 3 iterations	1141/218/42	11.3B/3.32B/1.40B
Sweep3D	DOE	$150 \times 150 \times 150$	2105/67/6	178B/53.4B/30.8B

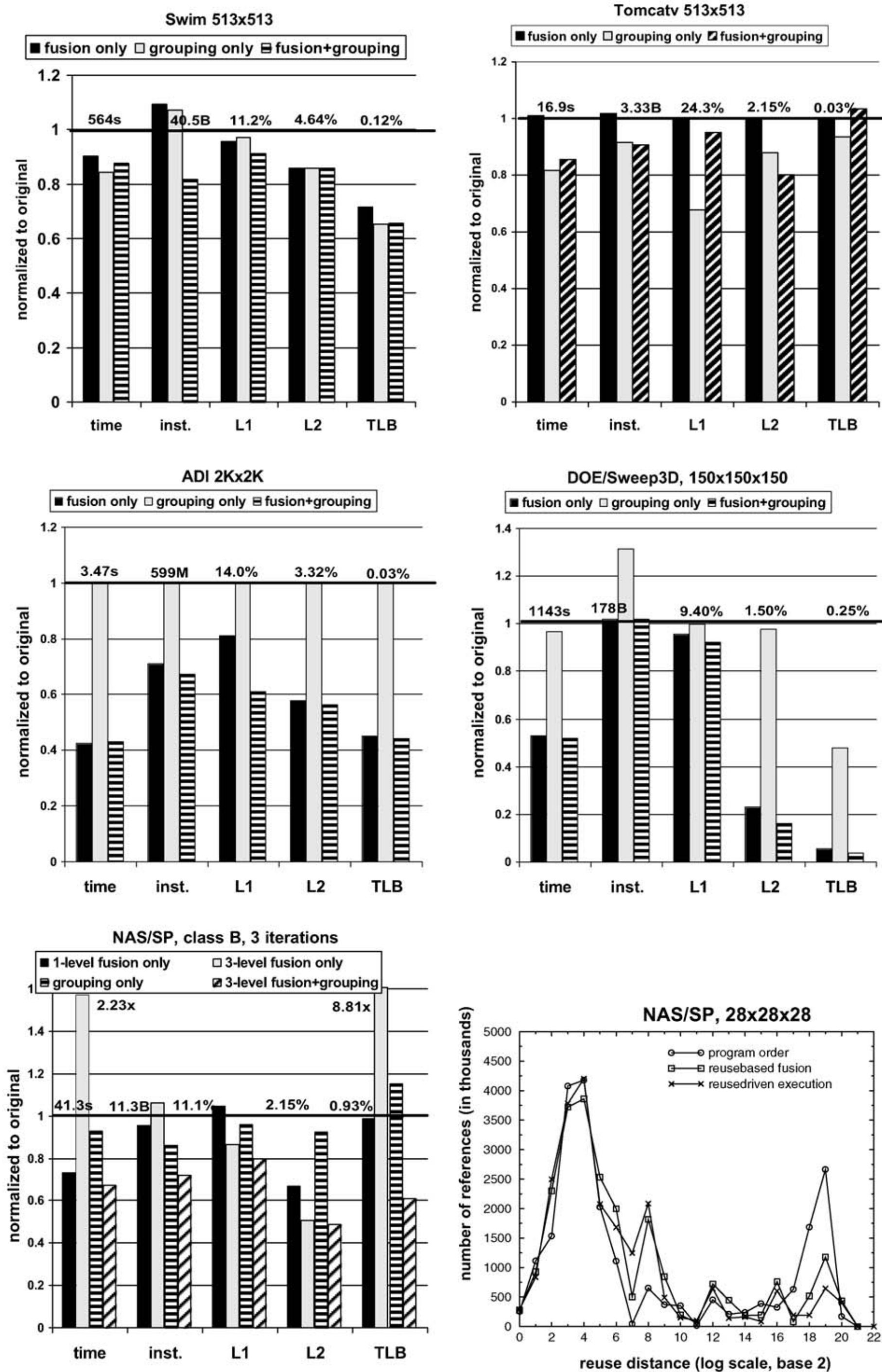


Fig. 10. The effect of transformations on SGI Origin 2000.

instructions. When combined, they execute 18% fewer instructions because they reduce memory access and simplify address calculation. However, the lower instruction count does not lead to significant performance gain.

Tomcatv has two pipelined computations progressing along reverse directions, so multi-level loop fusion fuses non-conflicting loops after interchanging them to the outside. Single-dimension data regrouping cannot find any opportunity in this program, but multi-level regrouping merges 7 arrays into 4. Loop fusion decreases performance by 1%, and the combined transformation reduces the execution time by 16%, mainly due to the 20% reduction in L2 misses. Data regrouping increases TLB misses by 3% because of the side effect of grouping at the outer data dimension. The increase is insignificant, considering that the TLB miss rate is only 0.03%.

In both *Swim* and *Tomcatv*, data regrouping performs slightly better without loop fusion. A likely reason is that the benefit of loop fusion does not outweigh its instruction overhead when the data size is small. The combined strategy does not lose on L2 cache performance: it removes similar L2 cache misses in *Swim* and 4% more in *Tomcatv*. Loop fusion should perform better with more optimized code generation or with larger inputs. We will use a much larger input size in the other three programs.

ADI performs pipelined computations in two dimensions. It uses the largest input size and incidentally gains the highest improvement. The reduction is 33% for instruction count, 39% for L1 misses, 44% for L2, and 56% for TLB. The execution time is reduced by 57%, a speedup of 2.33. Since only three arrays are used in the program, data regrouping has little benefit on L2, TLB, and the execution time, but it reduces L1 misses by 20%. Without loop fusion, data regrouping sees no opportunity and has no effect.

Sweep3D, the largest program in code size, also sees a dramatic improvement. It traverses three-dimensional arrays in six directions, including three along diagonals. The reduction is 84% in L2 and 96% in TLB. The overall performance is improved by a factor of 1.9. Fusion accounts for the most improvement, while array regrouping further improves performance by a small percentage. Without loop fusion, array regrouping reduces execution time by 3%, L2 misses by 2%, and TLB misses by 52% but it increases instruction count by 32%.

4.3.1. Program changes for SP

SP is a full application and deserves special attention in evaluating the global strategy. It simulates fluid dynamics by solving Navier–Stokes equation. The computation proceeds along three dimensions of data with many steps processing each dimension. The main computation subroutine, *adi*, has 67 nests after in-lining. Loop distribution and loop unrolling result in 482 loops at three levels—157 loops at the first level, 161 at the second, and 164 at the third. One-level loop fusion merges all level-one loops into 8 loop nests. The full fusion further merges loops in the remaining two levels and produces 13 loops at the second level and 17 at the third. Loop fusion increases the program size of the main computation routine from 1141 lines to 28805 lines, although the size increase of the executable is much smaller (from 490 to 526 kB).

The original program uses 15 global arrays. Array splitting results in 42 arrays. After full loop fusion, data regrouping combines 42 arrays into 17 new ones. The choice of regrouping is very different from the original ones coded by the programmer. For example, one new array consists of four original arrays: $\{ainv(N,N,N), us(N,N,N), qs(N,N,N), u(N,N,N,1-5)\}$, and another new array includes two disjoint sections of an original array: $\{lhs(N,N,N,6-8), lhs(N,N,N,11-13)\}$.

One-level fusion increases L1 misses by 5%, but reduces the instruction count by 4%, L2 misses by 33% and execution time by 27%, suggesting that the performance bottleneck is memory bandwidth. Fusing all levels eliminates half of the L2 misses (49%). However, it creates too much data access in the innermost loop and causes 8 times more TLB misses. The performance is slowed by a factor of 2.32. Data regrouping, however, merges data in affinity groups and achieves the best performance. It reduces the instruction count by 33%, L1 misses by 20%, L2 by 51%, and TLB by 39%. The execution time is shortened by one third (33%), a speedup of 1.5 (from 64.5 Mf/s to 96.2 Mf/s).

Without loop fusion, however, data regrouping can combine only two arrays, $lhs(N,N,N,7)$ and $lhs(N,N,N,8)$, which yields a modest improvement, reducing the instruction count by 7%, L1 misses by 4%, L2 misses by 8%, and execution time by 7%. It increases TLB misses by 15%.

Greedy fusion does not consider resource constraints such as the number of available registers. It may fuse too much code into the innermost loop and cause excessive register spilling. For *SP*, the original execution has 4.6 billion register loads and stores. One-level loop fusion and array regrouping reduce the number to 3.3 billion because they simplify address calculation and utilize integer registers better. After full fusion, however, the number of loads and stores is increased to 4.0 billion. We expect to avoid this increase by using a new algorithm for loop fusion that considers resource constraints.

We now compare the source-level fusion with the trace-level fusion described in Section 2.3. The bottom-right graph of Fig. 10 compares the effect of loop fusion with that of reuse-driven execution for *SP*. It shows the histogram of reuse distances for three versions of *SP*: the original program, reuse-based loop fusion, and reuse-driven execution. Reuse-based fusion reduces 45% long-distance reuses, which is not as good as the 63% reduction by trace-level reuse-driven execution. However, reuse-based loop fusion does realize a major portion of the potential. Furthermore, the reduction on long-distance reuses is very close to the reduction of L2 misses on Origin2000 (51% vs. 45%), indicating that the measurement of reuse distance closely matches L2 cache performance on SGI.

4.3.2. Effect on Sun and Compaq systems

We tested two other high-end systems: a 336 MHz UltraSparc-II processor on a Sun Enterprise server and a 600 MHz Alpha 21164 processor on a Compaq 4000 server. The UltraSparc-II processor does not have hardware performance counters. The ones on the Alpha

21164 are not turned on. So we could measure only the running times, which are listed in Table 4. For each program, we tested four versions: original (*orig*), fusion only (*fuse*), regrouping only (*grp.*), and combined optimization (*both*). The table lists execution times by seconds or speed by Mflop/s, as well as the relative speedup compared to the original version. We could not compile *Sweep3D* on the Sun machine because of the absence of a Fortran 90 compiler.

In general, the performance on the Sun processor is a factor of two to four slower than the performance on SGI. Sun's compiler does not implement as aggressive loop-nest optimization or prefetching as the SGI compiler does. Since programs run slower, the bandwidth constraint is less severe and the transformations are less effective. However, they improve performance in all programs, from 5% in *Tomcatv* to 20% in *NAS/SP* with an average of 13%. The numbers show the importance of a combined strategy. Applying loop fusion or data regrouping alone is not always beneficial. In each program, at least one of fusion and regrouping does not result in any improvement. In contrast, the combined strategy consistently improves performance in all programs. The most dramatic example is *NAS/SP*, for which fusion increases running time by 3%, regrouping increases it by 16%, but the combination reduces running time by 20%.

The performance of the Alpha processor is about 10% slower than SGI. The bandwidth constraint is significant at this performance level. The transformations are very effective. With the exception of *Tomcatv*, the new strategy improves performance by 17% to a factor of 2.4 with an average of 66%. The combination of loop fusion and regrouping gives the best performance in three programs. In the other two, the difference is small—3% in *Sweep3D* and 7% in *Tomcatv*. Due to the lack of performance tools, we did not investigate the cause for the slowdown.

The results on Sun and Compaq systems have shown that the global strategy is effective on these two types of machines. The combination of loop fusion and data regrouping is crucial in achieving consistent performance improvement. The Alpha processor sees the largest performance improvement, the whole-program performance of *NAS/SP* is increased by a factor of 2.4.

4.4. Affinity-based structure splitting

In this section, we evaluate affinity-based structure splitting and compare it with three other schemes: array organization, structure organization, and frequency-based structure splitting. The attributes of an object are stored in different arrays in the array organization and in the same structure in the structure organization. Frequency-based structure splitting was used by Chilimbi et al., who divided an object into two parts, one including frequently accessed attributes and the other including the rest [15].

We test two real applications in the experiment. The first, *Magi*, comes from the Phillips Lab of Department of Defense. It simulates high-impact particle dynamics through the smoothed particle hydrodynamics (SPH) method. It has over 9000 lines of Fortran 90 code. Each particle has 26 attributes, accessed in 6 program phases. A simplified view of the program was shown before in Table 2. The access pattern depends on the distribution of particles, and the access changes as particles move in space. Affinity-based regrouping groups 26 attributes into 6 arrays. Frequency-based splitting puts all 26 attributes in one structure, since all of them are frequently used. We use a user-supplied input of 28K particles and run the program on a 195 MHz MIPS R10K processor on SGI Origin 2000. The user specified the compiler flag *-O2* in order to preserve numerical accuracy. All our transformations preserve numerical accuracy because they change only the data layout.

Table 4
Performance on Sun Ultra-II and Compaq Alpha processors

Applications		336 MHz Ultra-II Enterprise 4500 server SUN Fortran 4.0				600 MHz Alpha 21164 Alpha 4000 server Compaq Fortran X5.3			
		Orig	Fuse	Grp.	Both	Orig	Fuse	Grp.	Both
Swim	time (sec.)	262	286	241	246	150	152	141	128
	speedup	1.00	0.92	1.09	1.07	1.00	0.99	1.06	1.17
Tom-catv	time (sec.)	71.6	68.4	71.6	68.1	18.5	19.7	18.5	19.8
	speedup	1.00	1.05	1.00	1.05	1.00	0.94	1.00	0.93
ADI	time (sec.)	13.7	11.0	13.7	11.5	6.71	3.44	6.71	3.17
	speedup	1.00	1.25	1.00	1.19	1.00	1.95	1.00	2.12
SP	Mflop/sec.	39.7	38.7	33.5	47.8	54.1	89.6	55.4	96.0
	speedup	1.00	0.97	0.84	1.20	1.00	2.26	1.40	2.42
Sweep-3D	time (sec.)	could not compile				1358	785	1565	813
	speedup	Fortran 90				1.00	1.73	0.87	1.67
Average speedup		1.00	1.05	0.98	1.13	1.00	1.38	1.07	1.66

The second test program, *Cheetah*, is a simulator for fully associative cache [53], widely distributed as part of SimpleScalar [39]. It has 718 lines in its main program file and 2287 lines counting all user-written header files. We do not have a C compiler, so program analysis and transformation are done by hand. The program uses a self-adjusting tree (splay tree), composed of dynamically allocated *tree_node* structures linked by pointers. Profiling analysis shows that two most time-consuming functions are *ref_tree* and *rotate_right*. A manual inspection reveals three phases with the following attribute access: *(inum, lft)*, *(inum, rtwt, addr, rt)*, and *(lft, rt, rtwt)*. Hence, for affinity-based regrouping, we have all attributes in separated arrays except for *rt* and *rtwt*. Profiling shows that *lft* and *rtwt* are used at least twice as frequently as other attributes. For frequency-based splitting, we have these two attributes in one structure and all others in another. The input to *Cheetah* is a loop that iterates two million data elements twice. The experiment was performed a few years after that of *Magi*, so we used a different system: a 250 MHz MIPS R10K processor on SGI Origin2000.

To apply structure splitting in *Cheetah*, we convert the program from using pointer reference to using array indexing. For example, we store left-child pointers in an array and convert their references, for example, from `p->lft` to `lft[pi]`. Since *Cheetah* allocates a fixed number of tree nodes, we can pack these tree nodes into an array. The conversion reduces execution time from 7.44 to 6.31 s. In comparison, Chilimbi et al., did not convert data to arrays [15]. Instead, they added a pointer to link the split pieces. Their scheme supports dynamic allocation, although the pointer adds a space and time cost. In the above experiment, however, we use the array code for all types of data layout including frequency-based splitting.

For each program and each layout, Table 5 shows the execution time and the number of cache and TLB misses, as measured by hardware counters. The initial data layouts—array layout in *Magi* and structure layout in *Cheetah*—happen to be the worst for performance. In contrast, affinity-based regrouping achieves the fastest running time and lowest number of misses in TLB and level-one cache. The number of misses in the level-two cache is also lowest in *Magi*, but it is marginally higher (1%) than array layout in *Cheetah*, likely due to non-uniform array sizes in the grouped layout. Despite this, it is clear that affinity-based regrouping gives the best data layout, improving program performance by 32% in *Magi* and 19% in *Cheetah*, compared to the initial data layout.

Frequency-based structure splitting improves the initial layout to a lesser degree—12% in *Magi* and 6% in *Cheetah*. The improvement from affinity is about three times as much as that from frequency in part because the former transfers 32% less memory data than the latter does. The reason that frequency-based splitting causes more memory transfer is that object attributes are frequently accessed but not frequently accessed together. Although not shown in the table, the experiment studies a few other choices of structure splitting but none performs as well as affinity-based structure splitting.

In the past, data optimization has been the responsibility of the programmer. Our results have shown that this is unnecessary and improper. Neither Fortran-style array layout nor C-style structure layout gives the best performance. The optimal layout depends on the reference affinity in a program, which may change if some parts of a program are modified. Automatic data regrouping would save the programmer from having to redo the data layout. It gives programmers complete freedom in defining data structures without worrying about their impact on performance. It gives the optimal data layout regardless of the initial choice or subsequent changes of a programmer.

Table 5
Comparison of affinity-based object reorganization with array layout, structure layout, and frequency-based structure splitting

Performance	Data layout			
on MIPS R10K	I. Array	II. Structure	Regrouping based on	
SGI Origin2000	layout	layout	III. frequency	IV. affinity
Magi, a hydrodynamics simulator				
Exe. Time (s)	885	787	same as II	673
Level-1 misses (billion)	5.64	3.67	same as II	3.49
Level-2 misses (million)	238	314	same as II	197
TLB misses (million)	1080	606	same as II	574
Cheetah, a fully-associative cache simulator				
Exe. Time (s)	5.37	6.31	5.96	5.30
Level-1 misses (million)	11.9	15.1	15.0	11.9
Level-2 misses (million)	2.87	4.81	4.27	2.90
TLB misses (thousand)	29.4	55.6	44.3	28.5

4.5. Comparison with traditional compiler optimization

We compare the new strategy with traditional compiler optimization implemented in the SGI compiler. Table 6 shows the amount of data transferred for versions of each program with optimizations provided by the SGI compiler and after transformation via the strategy developed in this paper. The results are normalized to the version with no optimization. If we compare the average *reduction* in misses due to compiler techniques, the new strategy, labeled by column *New*, does better than the SGI compiler by factors of 8 for L1 misses, 5 for L2 misses, 2.5 for TLB misses, and 1.6 for performance. In addition, the figures of the last two applications show that the SGI compiler is particularly ineffective in optimizing the two large applications, on which the new strategy works especially well. Thus, the global strategy we propose has a clear advantage over the more local strategies employed by an excellent commercial compiler, especially for large programs with huge data sets.

4.6. Summary

As a result of better global cache reuse, the new strategy consistently improves program performance on three different machine architectures over three different vendor compilers. With the exception of *Tomcatv* on the Alpha processor, the combined strategy improves whole-program performance in every case. The improvement is dramatic in the two fast processors—SGI R12K and Compaq Alpha 21164. The average improvement is a factor of 1.61 on the SGI system and 1.66 on the Compaq system.

The reduction in memory traffic and the improvement in execution time are obtained through source-to-source compiler techniques developed for reducing program bandwidth consumption. We highlight three features.

- Reuse-based loop fusion. Pair-wise fusion can fuse loops of different control structures and therefore find more fusion opportunities. In *NAS/SP*, over 100 loops is fused into a single loop.
- Affinity-based data regrouping. Data regrouping improves performance in almost all programs on all machines. Multidimensional regrouping is necessary to fully exploit reference affinity at all granularity, especially in fused loops, which are often imperfectly nested.
- Combined strategy. Loop fusion may degrade performance without data regrouping, and data regrouping may see less opportunity without loop fusion. However, when combined, they almost always achieve a dramatic improvement. *It is the combination that gives us the most effective strategy for global optimization.*

Table 6
Summary of effect on memory traffic, measured on SGI Origin 2000

Program	L1 misses		L2 misses		TLB misses		Speedup
	SGI	New	SGI	New	SGI	New	over SGI
Swim	1.26	1.15	1.10	0.94	1.60	1.05	1.14
Tomcatv	1.02	0.97	0.49	0.39	0.010	0.010	1.17
ADI	0.66	0.40	0.94	0.53	0.011	0.005	2.33
Sweep3D	1.00	0.92	0.99	0.16	1.00	0.04	1.93
NAS/SP	0.97	0.77	1.00	0.49	1.09	0.67	1.49
Average	0.98	0.84	0.90	0.50	0.74	0.35	1.61

5. Related work

5.1. Related work in loop fusion

Many researchers have studied loop fusion. Early work included those of Wolfe [56] and Allen and Kennedy [4]. Combining loop fusion with distribution was originally discussed by Allen et al. [3]. They also used loop alignment to assist parallelization. To improve the reuse of vector registers, Allen and Kennedy fused loops with identical bounds, no fusion-preventing dependences, and no true dependences with intervening statements. Callahan developed a greedy fusion that maximizes coarse-grain parallelism but not locality [12]. McKinley et al. [43] implemented loop fusion for cache reuse. Both of these works used the same fusion constraints as Allen and Kennedy. McKinley et al. successfully fused on average 6% of tested loops. Fusion-preventing dependences can be avoided by *peel-and-jam* as shown by Porterfield [47]. Peel-and-jam is functionally the same as loop alignment but is restricted to a single direction. Manjikian and Abdelrahman used *peel-and-jam* to fuse loops with different iteration bounds but with the same control structure in the fused levels [41]. They found more opportunities for fusion. Still, at most 8 original loops could be fused into a single loop. Our pair-wise fusion can fuse loops of different control structures at all loop levels. To the best of our knowledge, this work finds the largest degree of fusion among published studies. Approximately 500 loops in *NAS/SP* were fused into 8 loop nests.

Loop fusion may cause poor spatial locality due to the increased data access in fused loops. McKinley et al. reported that fusion improved hit rate in four programs but made it worse in another three programs [43]. Manjikian and Abdelrahman used a form of array padding [41]. They did not address the situation when different loops require different data layouts.

Global loop fusion was formulated as a graph problem by Callahan for parallelization [12], by Gao et al. [24] for register reuse, and by Kennedy and

McKinley [31] for locality and parallelism. Kennedy and McKinley proved that optimal fusion is NP-hard [31]. All these studies modeled loops as nodes and data reuse as weighted edges. We use loop fusion to reduce program bandwidth consumption. We model data reuse with hyper edges that can connect to any number of loops. We prove that the extended problem is also NP-hard. We use a greedy heuristic that examines loops in program order. Kennedy gave a fast algorithm that always fuses along the heaviest edge and supports both models of reuse [28]. Graph-based methods have more freedom than our sequential scheme. However, regardless of the fusion heuristic, a global method needs to fuse loops of different control structures and to optimize data layout after fusion. Therefore, they should benefit from pair-wise fusion and data regrouping.

Gao et al. combined loop fusion with array contraction [24]. A similar strategy was used by Lin et al. using language support [37], by Lim et al. using affine partitioning [38], and by Song et al. [52] using a similar fusion scheme as Manjikian and Abdelrahman [41]. Lim et al. considered locality optimization in conjunction with parallelization. Our work complements the above studies: pair-wise fusion enables more fusion, and affinity-based array regrouping improves data layout for arrays that cannot be eliminated by contraction.

Kodukula et al. took a data-centric approach by fusing computations on each data tile through "shackling" [34]. Pugh and Rosser fused computations on each data element by slicing loop iterations [49]. Yi et al. organized computations in a recursive decomposition [59]. Two simultaneous studies by Song and Li [51] and by Wonnacott [57,58] tiled a time-step loop when its inner loops were all fusible. Loop tiling and fusion are complementary techniques—tiling brings together data reuses in the outer loops of a single loop nest, while loop fusion brings together data reuses from disjoint loops. Since tiling needs all inner loops fusible, it can use reuse-based loop fusion to handle inner loops of different shapes. When not all loops are fusible, tiling cannot be applied across disjoint loop nests. In this case, we show that optimal fusion is NP-hard and present a greedy heuristic. Several previous studies combined loop tiling and fusion using powerful but expensive analysis that computes all-to-all data dependence [34,49,59]. They often generate code that has higher instruction overhead. In comparison, reuse-based fusion is incremental and therefore can more efficiently reorganize a large number of loops. Pugh and Rosser showed mixed results for iteration slicing. On SGI Octane, *Swim* was improved by 10% but the transformation on *Tomcatv* "interacted poorly with the SGI compiler" [49]. Using different machines, Song and Li [51] and Wonnacott [58] achieved integer-factor performance improvement for *Swim* and *Tomcatv* by combining fusion with time-step tiling and single-array data transformations. They did not report the direct effect of loop fusion. Tiling is not applicable to the two large programs in our test suite, *SP* and *Sweep3D*, whose time steps include loops that are not all fusible. The second technique in this work, array regrouping, analyzes the access pattern in all program loops and improves spatial reuse in fused loops as well as tiled loops.

The importance of global data reuse was shown by McKinley and Temam, who observed that most misses in SPECfp and Perfect benchmarks are due to inter-nest temporal reuse [44]. Our reuse-driven execution complements their results by showing how much reuse can be converted to cache reuse by reordering.

Scholz developed a functional array language, *Single-Assignment C (SaC)*, and used loop fusion as an important optimization [50]. Since SaC programs have no side effects, loop fusion is greatly simplified: it does not need loop alignment for correctness. It can also effect array contraction through forward substitution. Scholz did not consider data transformation after loop fusion. Recently, Pingali et al. showed that computation fusion can be applied to non-scientific programs [46].

5.2. Related work in improving cache spatial locality

Loop interchange can make array access contiguous, as shown by Abu-Sufah et al. [1], Gannon et al. [23], Wolf and Lam [55], Ferrante et al. [22], and McKinley et al. [43]. Alternatively, Mace [40] and Leung [36] transformed array layout to effect contiguous access. Cierniak and Li [16] and Kendemir et al. [27] showed that loop and data transformations can be combined to achieve a greater effect. In the context of parallelization, Kennedy and Kremer [30], Anderson et al. [6], and Eggers and Jeremiassen [26] used data transformation to improve locality for parallel programs. Beckman and Kelly studied run-time data layout selection in a parallel library [9]. The goal of most of these techniques is to improve data reuse within a single array. An exception is *group & transpose*, which groups single-dimension vectors used by a thread to reduce false sharing [26]. Grouping all local data may reduce cache spatial locality if they are not used at the same time. In comparison, we group data based on reference affinity and group high-dimensional arrays at multiple granularity.

Locality between multiple arrays can be improved by array padding as reported by Bailey [8], which makes arrays well separated, and array copying as used by Lam et al. [35], which combines array sections at run time. Compared to array padding, data regrouping is preferable because it works for all sizes of arrays or all configurations of cache, but padding is still needed if not all arrays can be grouped together. Copying is not necessary if reference affinity exists. In the case of dynamic reference affinity, copying is needed. We have

formulated the problem in terms of reference affinity and analyzed its complexity.

5.3. Comparison with other locality models

For loop-based code, a compiler can construct very accurate locality models. For other programs, however, people have often relied on profiling. The concept of reference affinity can be used in both cases to denote data that are often accessed together. We now discuss the differences between reference affinity and other models used in profiling analysis.

Early analysis of execution frequency included counter-based profiling by Knuth [33] and static probability analysis by Cocke and Kennedy [17]. Access frequency does not distinguish the time of access. Being frequently accessed does not mean frequently accessed *together*. In the early 1980s, Thabit measured how often two data elements were used together [54]. Pair-wise affinity does not imply reference affinity. For example, the sequences *ab..ab bc..bc ca..ca* and *abc..abc* can produce the same pair-wise graph, but they have very different reference affinity. In a recent paper, Petrank and Rawitz formalized this observation and proved a harsh bound: with only pair-wise information, no algorithm can find a static data layout that is always within a factor of $k-3$ from the optimal solution, where k is proportional to the size of cache [45].

Chilimbi described a "hot-stream" as a model of locality [14]. A hot stream is an identical sequence of data accesses that occurs repeatedly in a program. By definition and by implementation (using grammar compression), hot-streams must be identical sub-sequences. It is different from reference affinity. For example, *a* and *b* are in an affinity group in the sequence *axb…ayb..azb*, but they are not part of a hot stream. Ding and Zhong recently measured data reuse pattern by the distribution of the distance of data reuses [21]. Reuse distance pattern is a summary over all reuses of all data and does not directly find reference affinity among data subsets.

6. Conclusion

This research has presented a two-step strategy for minimizing program bandwidth consumption. The first, *reuse-based loop fusion*, fuses loops of different control structures by pair-wise and sequential greedy fusion. The second step, *affinity-based data regrouping* groups data at all granularity to effect a compile-time optimal layout. Together, the two steps transform both program and data for the whole program and at fine granularity.

When evaluated on five standard benchmarks from SPEC, NASA, and DOE, the new strategy achieved significant reduction in both the amount of memory traffic and the program running time, outperforming current commercial compilers from SGI, Sun, and Compaq by integer factors for applications that consist of thousands lines of code. The evaluation also showed that the combination of computation fusion and data regrouping is crucial to realizing the benefit of the global strategy.

Broadly speaking, this work is a significant step toward alleviating the tension between today's software and hardware. Given the rapid innovation and diversification of computing devices, a major challenge is to adapt software to its hardware environment. In this work, we have demonstrated the benefit of reorganizing the whole program and its entire data set. If these strategies are incorporated into widely-used compilers it will make it easier for programmers to achieve high performance on leading-edge computer systems without the necessity of restructuring by hand.

Acknowledgments

The implementation of the compiler is based on the D System, a project led by John Mellor-Crummey and in part by Vikram Adve at Rice University. It was based on a scalar compiler framework put together by Nat Macintosh. Leo Boyarsky hand transformed the *Cheetah* program and measured the effect of data regrouping. We also thank Keith Cooper, Nathaniel Dean, Alan Cox, Kathryn McKinley, Larry Carter, Sarita Adve, Guang Gao, David Whalley, David Wonnacott, Uli Kremer, Kath Knobe, Matthias Felleisen, Shriram Krishnamurthi, Danny Sorensen, William Cook, and anonymous referees of this journal and of IPDPS'01 and IPDPS'00 conferences and LCPC'99 workshop for helpful comments at various stages of this work.

The work was supported by the Defense Advanced Research Projects Agency (Contract F30602-96-1-0159), a gift from Compaq Computer Corporation (Agreement US-1998057), and by the Department of Energy through the Los Alamos Computer Science Institute (Contract 74837-001-0349 from the Regents of University of California) and the Lawrence Livermore ASCI program (Contract B347884). Chen Ding is also supported by a Young Investigator grant from the Department of Energy (Contract DE-FG02-02ER25525). The machines were purchased by grants from the Defense Advanced Research Projects Agency (Contract 03891-001-99-49), Compaq Computer Corporation, and the National Science Foundation (Contract SCREMS-98-72009 and EIA-0080124).

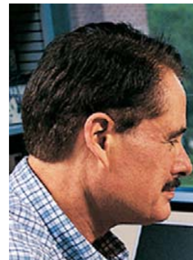
References

[1] W. Abu-Sufah, D. Kuck, D. Lawrie, On the performance enhancement of paging systems through program analysis and

transformations, IEEE Trans. Comput. C 30 (5) (May 1981) 341–356.
[2] V. Adve, J. Mellor-Crummey, Using integer sets for data-parallel program analysis and optimization, in: Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998.
[3] J.R. Allen, D. Callahan, K. Kennedy, Automatic decomposition of scientific programs for parallel execution, in: Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages, Munich, Germany, January 1987.
[4] J.R. Allen, K. Kennedy, Vector register allocation, IEEE Trans. Comput. 41 (10) (October 1992) 1290–1317.
[5] R. Allen, K. Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufman, Los Altos, CA, 2001.
[6] J. Anderson, S. Amarasinghe, M. Lam, Data and computation transformation for multiprocessors, in: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Santa Barbara, CA, July 1995.
[7] J. Backus, The history of Fortran I, II, and III, in: Wexelblat (Ed.), History of Programming Languages, Academic Press, New York, 1981, pp. 25–45.
[8] D. Bailey, Unfavorable strides in cache memory systems, Technical Report RNR-92-015, NASA Ames Research Center, 1992.
[9] O. Beckmann, P.H.J. Kelly, Efficient interprocedural data placement optimisation in a parallel library, in: Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, May 1998.
[10] L.A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Systems J. 5 (2) (1966) 78–101.
[11] D.C. Burger, J.R. Goodman, A. Kagi, Memory bandwidth limitations of future microprocessors, in: Proceedings of the 23th International Symposium on Computer Architecture, Philadelphia, PA, May 1996.
[12] D. Callahan, A Global Approach to Detection of Parallelism, Ph.D. Thesis, Dept. of Computer Science, Rice University, March 1987.
[13] D. Callahan, J. Cocke, K. Kennedy, Estimating interlock and improving balance for pipelined machines, Journal of Parallel and Distributed Computing 5 (4) (August 1988) 334–358.
[14] T.M. Chilimbi, Efficient representations and abstractions for quantifying and exploiting data reference locality, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Snowbird, UT, 2001.
[15] T.M. Chilimbi, B. Davidson, J.R. Larus, Cache-conscious structure definition, in: Proceedings of SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, GA, May 1999.
[16] M. Cierniak, W. Li, Unifying data and control transformations for distributed shared-memory machines, in: Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, CA, June 1995.
[17] J. Cocke, K. Kennedy, Profitability computations on program flow graphs, Technical Report RC 5123, IBM, 1974.
[18] E. Dahlhaus, D.S. Johnson, C.H. Papadimitriou, P.D. Seymour, M. Yannakakis, The complexity of multiway cuts, in: Proceedings of the 24th Annual ACM Symposium on the Theory of Computing, May 1992.
[19] C. Ding, Improving Effective Bandwidth through Compiler Enhancement of Global and Dynamic Cache Reuse, Ph.D. Thesis, Dept. of Computer Science, Rice University, January 2000.
[20] C. Ding, K. Kennedy, Memory bandwidth bottleneck and its amelioration by a compiler, in: Proceedings of International Parallel and Distributed Processing Symposium, Cancun, Mexico, May 2000.
[21] C. Ding, Y. Zhong, Predicting whole-program locality using reuse-distance analysis, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, CA, June 2003.
[22] J. Ferrante, V. Sarkar, W. Thrash, On estimating and enhancing cache effectiveness, in: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing, Fourth International Workshop, Springer, Santa Clara, CA, August 1991.
[23] D. Gannon, W. Jalby, K. Gallivan, Strategies for cache and local memory management by global program transformation, J. Parallel Distrib. Comput. 5 (5) (October 1988) 587–616.
[24] G. Gao, R. Olsen, V. Sarkar, R. Thekkath, Collective loop fusion for array contraction, in: Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, New Haven, CT, August 1992.
[25] P. Havlak, K. Kennedy, An implementation of interprocedural bounded regular section analysis, IEEE Trans. Parallel Distrib. Systems 2 (3) (July 1991) 350–360.
[26] T.E. Jeremiassen, S.J. Eggers, Reducing false sharing on shared memory multiprocessors through compile time data transformations, in: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Santa Barbara, CA, July 1995, pp. 179–188.
[27] M. Kandemir, A. Choudhary, J. Ramanujam, P. Banerjee, A matrix-based approach to the global locality optimization problem, in: Proceedings of International Conference on Parallel Architectures and Compilation Techniques, 1998.
[28] K. Kennedy, Fast greedy weighted fusion, in: Proceedings of the 2000 International Conference on Supercomputing, Santa Fe, NM, May 2000.
[29] K. Kennedy, C. Ding, Resource constrained loop fusion, Technical Report TR03-424, Department of Computer Science, Rice University, September 2003.
[30] K. Kennedy, U. Kremer, Automatic data layout for distributed memory machines, ACM Trans. Programming Languages Systems (TOPLAS) 20 (4) (1998).
[31] K. Kennedy, K.S. McKinley, Typed fusion with applications to parallel and sequential code generation, Technical Report TR93-208, Dept. of Computer Science, Rice University, August 1993 (also available as CRPC-TR94370).
[32] D.G. Kirkpatrick, P. Hell, On the completeness of a generalized matching problem, in: The Tenth Annual ACM Symposium on Theory of Computing, 1978.
[33] D. Knuth, An empirical study of FORTRAN programs, Software—Practice Experience 1 (1971) 105–133.
[34] I. Kodukula, N. Ahmed, K. Pingali, Data-centric multi-level blocking, in: Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation, Las Vegas, NV, June 1997.
[35] M. Lam, E. Rothberg, M.E. Wolf, The cache performance and optimizations of blocked algorithms, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), Santa Clara, CA, April 1991.
[36] S. Leung, Array restructuring for cache locality, Ph.D. Thesis, Technical Report UW-CSE-96-08-01, University of Washington, 1996.
[37] E. Lewis, C. Lin, L. Snyder, The implementation and evaluation of fusion and contraction in array languages, in: Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998.
[38] A. Lim, S. Liao, M. Lam, Design and evaluation of locality optimizations using affine partitioning, in: Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Snowbird, UT, June 2001.

[39] SimpleScalar LLC, SimpleScalar tool set, www.simplescalar.com.
[40] M.E. Mace, Memory storage patterns in parallel processing, Kluwer Academic, Boston, 1987.
[41] N. Manjikian, T. Abdelrahman, Fusion of loops for parallelism and locality, IEEE Trans. Parallel Distrib. Systems 8 (1997).
[42] R.L. Mattson, J. Gecsei, D. Slutz, I.L. Traiger, Evaluation techniques for storage hierarchies, IBM System J. 9 (2) (1970) 78–117.
[43] K.S. McKinley, S. Carr, C.-W. Tseng, Improving data locality with loop transformations, ACM Trans. Programming Languages Systems 18 (4) (July 1996) 424–453.
[44] K.S. McKinley, O. Temam, Quantifying loop nest locality using SPEC'95 and the perfect benchmarks, ACM Transactions on Computer Systems 17 (4) (November 1999) 288–336.
[45] E. Petrank, D. Rawitz, The hardness of cache conscious data placement, in: Proceedings of ACM Symposium on Principles of Programming Languages, Portland, OR, January 2002.
[46] V.K. Pingali, S.A. McKee, W.C. Hsieh, J.B. Carter, Computation regrouping: restructuring programs for temporal data cache locality, in: Proceedings of ACM International Conference on SuperComputing, 2002.
[47] A. Porterfield, Software methods for improvement of cache performance, Ph.D. Thesis, Dept. of Computer Science, Rice University, May 1989.
[48] W. Pugh, A practical algorithm for exact array dependence analysis, Comm. ACM 35 (8) (August 1992) 102–114.
[49] W. Pugh, E. Rosser, Iteration space slicing for locality, in: Proceedings of the Twelfth Workshop on Languages and Compilers for Parallel Computing, August 1999.
[50] S. Scholz, With-Loop-Folding in SAC - condensing consecutive array operations, in: Proceedings of Conference on Implementation of Functional Languages, 1997.
[51] Y. Song, Z. Li, New tiling techniques to improve cache temporal locality, in: Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation, Atlanta, GA, May 1999.
[52] Y. Song, R. Xu, C. Wang, Z. Li, Data locality enhancement by memory reduction, in: Proceedings of ACM International Conference on Supercomputing, June 2001.
[53] R.A. Sugumar, S.G. Abraham, Multi-configuration simulation algorithms for the evaluation of computer architecture designs, Technical Report, University of Michigan, 1993.
[54] K.O. Thabit, Cache Management by the Compiler. Ph.D. Thesis, Dept. of Computer Science, Rice University, 1981.
[55] M.E. Wolf, M. Lam, A data locality optimizing algorithm, in: Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Canada, June 1991.
[56] M.J. Wolfe, Optimizing Supercompilers for Supercomputers. Ph.D. Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.
[57] D. Wonnacott, Using time skewing to eliminate idle time due to memory bandwidth and network limitations, in: Proceedings of International Parallel and Distributed Processing Symposium, Cancun, Mexico, May 2000.
[58] D. Wonnacott, Achieving scalable locality with time skewing, Internat. J. Parallel Programming 30 (3) (June 2002).
[59] Q. Yi, V. Adve, K. Kennedy, Transforming loops to recursion for multi-level memory hierarchies, in: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000.
[60] Y. Zhong, C. Ding, K. Kennedy, Reuse distance analysis for scientific programs, in: Proceedings of Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers, Washington DC, March 2002.

Chen Ding is an Assistant Professor of Computer Science at University of Rochester. His research focuses on understanding whether a complex program has an inherent pattern of data access and, if so, to what degree that pattern can be modeled, measured, and modified (improved). He is a recipient of a Young Investigator Award from the Office of Science of the US Department of Energy, a Career Award from the National Science Foundation, and a best paper award from the IEEE IPDPS'01 conference. In 2002, he co-organized the first ACM SIGPLAN Workshop on Memory System Performance.



Ken Kennedy is the John and Ann Doerr University Professor of Computer Science and Director of the Center for High Performance Software Research (HiPerSoft) at Rice University. He has supervised 35 Ph.D. dissertations and published two books and over 170 technical articles on compilers and programming support software for high-performance computer systems. In recognition of his contributions to software for high-performance computation, he received the 1995 W. Wallace McDowell Award, the highest research award of the IEEE Computer Society. In 1999, he was named the third recipient of the ACM SIGPLAN Programming Languages Achievement Award.

CN 43-1258/TP
ISSN 1007-130X

计算机工程与科学
Computer Engineering & Science

第36卷第1期2014年1月
Vol. 36, No. 1, Jan. 2014

文章编号:1007-130X(2014)01-0001-05

多核程序交互理论及应用*

丁 晨[1],袁 良[2]

(1. 罗切斯特大学计算机系,纽约 罗切斯特 14627,美国;2. 中科院计算所计算机体系结构国家重点实验室,北京 100190)

摘 要:多核处理器上共享缓存使用效率,即程序局部性是影响并行程序性能的关键因素之一。提出了以足迹为基础的局部性理论。介绍了缺失率、重用距离和足迹之间的转化关系,并利用足迹可组合性特征建立了并行程序局部性预测模型。

关键词:局部性;足迹;重用距离;缺失率

中图分类号:TP311.5　　**文献标志码**:A

doi:10.3969/j.issn.1007-130X.2014.01.001

Program interaction on multicore: Theory and applications

DING Chen[1], YUAN Liang[2]

(1. Department of Computer Science, University of Rochester, Rochester 14627, USA;
2. SKL of Computer Architecture, Institute of Computing Technology, Chinese Academy of Science, Beijing 100190, China)

Abstract: On modern multicore systems, the interaction between co-run programs largely depends on cache sharing, and cache sharing depends on the locality, i. e. the active data usage, in co-run programs. The paper introduces a newly developed locality theory based on a concept called the footprint. The theory shows the composability of footprint and the conversion between footprint and other locality metrics including the miss rate and reuse distance. The new theory enables accurate analysis of performance and dynamic cache allocation in the shared cache on multicore processors.

Key words: locality; footprint; reuse distance; miss ratio

1 引言

在科学研究、工程设计、商业处理甚至日常生活中,计算是无处不在的。由于硬件的发展,目前大多数计算、云、桌面和手持电脑程序在多核处理器上运行,同时运行的程序间会由于共享资源而产生相互作用和干扰,因此共享环境下的一个重要问题是如何控制程序间的交互和减少干扰影响。其目的不只是提高程序性能,还需要保证性能的稳定性。并且不只对并行程序,对并行运行的多个串行程序也同样重要。

共享缓存是程序干扰的首要原因。现代应用程序大部分运行时间开销为对内存数据的访问。通常情况下超过99%的数据读写操作发生在高速缓存。一个计算系统通常有2~8个处理器(Socket),每个处理器有2~6个物理核,每个物理核有2~4个超线程或称逻辑核,因此近百个任务可以一起并行运行。

解决程序干扰的一个途径是高速缓存分区,通过隔离程序解决干扰问题。然而,高速缓存分区是一种浪费,当只有一个或少数程序执行时会产生大量闲置缓存空间。当多线程程序共享数据时,同一数据需要多处存放。当前多核处理器配置多层私有和共享缓存。例如,Intel Nehalem 处理器拥有256 KB L2 私有缓存,和4 MB~8 MB L3 共享缓存。IBM 的 Power 7 拥有8个内核,256 KB 的 L2 私有缓存和32 MB L3 共享缓存。

* 收稿日期:2013-10-30;修回日期:2013-12-15
基金项目:国家自然科学基金资助项目(61133005,61272136,61221062,61328201)
通信地址:14627 美国纽约州罗切斯特市罗切斯特大学计算机系
Address: Department of Computer Science, University of Rochester, Rochester 14627, New York, USA

共享缓存的程序以不同的方式相互影响。例如,物理内核私有一级和二级缓存,但共享最后一级缓存;逻辑内核共享所有层次缓存;不同的处理器不共享缓存;然而所有程序共享内存带宽,而带宽需求又完全取决于缓存效率。此外,某些缓存策略,例如,Intel 的包容缓存有可能诱发间接影响,例如一个程序可能由于其他程序对共享缓存数据的访问而丢失自身私有缓存数据。

共享缓存的问世让人联想到早期分时计算机发明的共享内存。内存管理问题已被深入研究并得到了较好的解决,现代操作系统为大量程序分配和管理内存。然而,高速缓存由硬件而不是操作系统来管理,因此缓存共享问题更为复杂。缓存有多个层次、多种不同私有和共享组合方式,缓存的数据读写次数相比内存有更高数量级。一个程序可以每秒访问缓存 10 亿次,可以在不到一毫秒替换整个缓存内容。多个程序并行运行时的访问强度更是单一程序的数倍。此外,高速缓存的大小是固定的,除更换机器外,不能增置更多的缓存。

高速缓存中的程序干扰有不对称性、非线性和反馈性。文献[1]的实验结果展示了不对称性,这一性质也在更多的研究中被证实。在程序配对运行实验中,我们使用了文献[1]的设置,发现一个程序性能降低 85%,而其并行执行的伙伴程序性能只降低了 15%。因此,程序相互干扰影响大小不仅取决于并行运行的程序数目,还取决于程序的特性和类型。其次,干扰具有自反馈,因为一个程序影响并行程序,而并行程序又影响本身。

局部性是一个计算系统的基本属性。文献[2]提出如下概念:一个程序运行时的每一阶段只使用所有数据的一个子集。数据和数据是有区别的,所需要的数据不等于正在使用的数据,后者是前者的一部分。文献[3]命名了后来广泛使用的术语工作集,即为正在使用的数据子集。这些数据也可以称为活跃数据。计算机存储系统的性能取决于如何快速存取活跃数据,而对其他数据的访问时间无关紧要。存储研究必须将局部性这一概念量化。科学和工程研究的首要任务是度量,同样存储系统设计也离不开对活跃数据的定义和度量,因为“没有度量就没有提高”,而局部性分析正是对活跃数据的量化。

本文使用一个称之为程序足迹工作集的概念来度量活跃数据。我们定义数据足迹为程序在一个时间窗口内访问的数据量。足迹就是数据脚印的统计平均。

自然来说,窗口越长,访问数据越多,脚印也越大。脚印的大小显示各个时期活跃数据的多少。

作为一个例子,使用单词“footprint”作为一个数据序列。7 个字母的 9 次出现可看作对 7 个数据块的 9 次访问。每个连续子串是一个时间窗口。它的脚印是子串中不同字母的个数。

如图 1 所示,长度为 9 的有 45 个不同的时间窗口($C_9^2|9$),相应地产生 45 个脚印。随着窗口长度(x 轴)从 1 增加至 9,脚印(y 轴)从 1 增加到 7。最短的窗口长度是 1,有 9 个这样的窗口,其脚印大小自然是 1。最长的窗口就是整个序列,只有一个这样的窗口,长度是 9,脚印是 7,这也是整个序列的脚印。

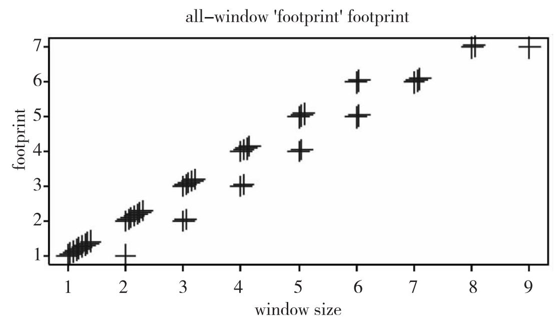


Figure 1 The footprint of “footprint”

图 1 “footprint”的足迹

在实际程序中,脚印数量为运行时间的二次方函数(运行时间为 N,窗户的数量(脚印个数)是 C_N^2)。假设一个程序在 3 GHz 处理器上运行 10 秒,可以得到 3E10 CPU 周期和 4.5E20 个不同窗口。

文献[4]指出,程序分析是一个大数据问题,并且展示了该问题规模和窗口数量的关系。图 2 显示脚印分析的问题规模。当程序运行时间从 1 秒增加至 1 个月,CPU 周期数从 3E9 增加至 2E15,不同的执行窗口从 4.5E18 增加至 5.8E29,也就是说,从 4 百亿亿增加至半个亿亿亿亿(nonillion)。

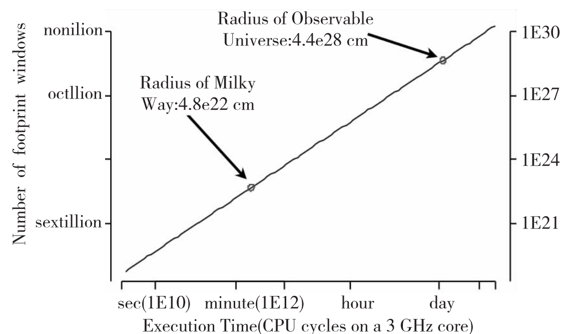


Figure 2 CPU execution time vs number of windows

图 2 CPU 执行时间与相应足迹窗口数量

作为一个动态问题，脚印分析的规模轻而易举赶超任何静态问题的规模。作为比较，图 2 中显示了银河系的半径 480 万亿亿厘米，可观测宇宙的半径 4 万亿亿亿厘米。

程序足迹理论的目的，首先就是要降低分析的难度，刻画所有窗口的活跃数据使用，并使其可用于系统的分析和优化。

2　程序足迹理论

局部性分析的基本单位是一个数据访问，它们的基本关系是数据重用。局部性理论是关于数据访问和重复使用的基本规律和性质，正如图论是关于节点和链接的基本规律和性质。程序足迹理论是一套新的局部性理论，它以数据脚印为核心，包括一系列概念、算法和定理。本节介绍新理论的四个组成部分和配套技术。

2.1　足迹测量

足迹就是数据脚印的统计平均。新理论给出足迹测量的三个算法。每一个比前一个在效率上高两个数量级。

(1)足迹分布分析用 $O(N\log M)$ 的时间列举所有 $O(N^2)$ 的数据脚印，其中 N 为运行长度，M 为最大脚印。这一算法发表在 2011 年 PPOPP 会议[5]。

(2)平均足迹分析用线性时间的成本 $O(n)$ 计算平均脚印大小，而不必枚举所有脚印。这一算法发表于 2011 年 PACT 会议[7]。

(3)足迹采样分析以有限大小的窗口为样本，更加降低了分析代价。这一算法发表于 2013 年 ASPLOS 会议[8]。

第一个算法测量所有窗口的脚印。因为它枚举所有的脚印，对每个窗口长度，它将找到最大、最小、中值和任何分布百分比的脚印。然而，分析代价有时是数千次放缓。

第二个算法只计算平均脚印，将成本从一千倍放缓降低至约 20 倍。作为一个线性时间的算法，它的代价和目标程序的长度成比例，因此是可扩展的分析。

真机上的高速缓存大小有限，所以分析上不必考虑所有窗口大小。运行时间长的程序行为往往重演，所以分析可以抽样。此外，在多核处理器上，分析可以单独使用一个核并行执行。第三个算法使用足迹采样和并行分析，将平均成本降低到 0.5%的运行时间。

2.2　组织协同关系模型

相比其他局部性指标，足迹有一个独特的性质：它是可组合的。令一个程序在长度为 X 的窗口中的平均足迹为 $prog.fp(X)$。如果有 K 个程序，$prog_1$、$prog_2$、…、$prog_K$ 共享缓存，则平均足迹是个别足迹的和：

$$corun.fp(X) = \sum_{i=1}^{K} prog_i.fp(X)$$

相比较而言，缓存缺失率不可组合。例如两个程序共享缓存。因为现在每个程序只是使用一部分缓存，而不是整个缓存，并行运行程序的缺失将增加，总缺失数将大于单独运行下的缺失数之和。缺失率的变化，正如前面提到，是不对称的、非线性的，并且受自反馈。因此，我们不能直接用单独运行程序的缺失率推断并行运行程序的缺失率。另一种局部性指标是重用距离。重用距离不依赖于具体缓存参数，但也不可组合。

接下来的问题是，可组合的足迹是否能帮助分析缺失率、重用距离和其他局部性指标。新理论的第三部分将解决这个问题。

2.3　局部性的度量转换

足迹理论给出了度量转换的换算公式[7]。例如，下面的公式用足迹计算出缺失率：

$$mr(C) = \frac{fp(X+\triangle X) - fp(X)}{\triangle X}$$

缓存大小 $C=fp(X)$。从函数概念来说，缺失率是足迹的导函数。足迹理论证明了足迹是个凹函数，所以推导出的缺失率是单调非递减。度量转换是可逆的。如果我们有所有缓存的命中率大小，我们可以逆转公式和计算平均足迹。逆过程通过离散函数的叠加，类似连续函数的积分。

如果将足迹组合和度量转换依次使用，我们可以看到，从单独运行程序的足迹可以推导出并行运行程序的缺失率。第一，计算并行运行程序的足迹。第二，计算出并行运行程序的缺失率。图 3 显示了推导关系。

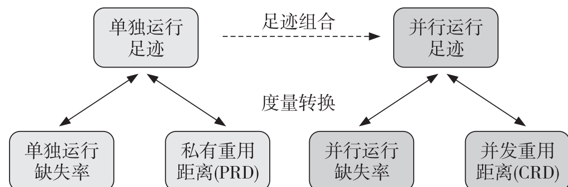


Figure 3　Joint use of two theoretical properties: composition (dotted line) and conversion (solid lines)

图 3　两种理论特性，组合关系(虚线)和转换关系(实线)的联合使用

由于转换公式是可逆转的,因此可以在足迹和缺失率之间互推,分析的功能大大增强。其一个效果是,我们可以间接组合缺失率。首先,用每个程序的缺失率计算其单独运行的足迹;然后累加每个程序的脚印,计算出并行运行程序足迹。最后转换出并行运行程序缺失率。这类推导可以间接组合其它局部性指标,比如重用距离。组合私有重用距离可以推算出并发重用距离[8]。

合二而一之后还需要一分为二。在组合问题解决之后,我们可以着手下一个问题:分解问题。并行运行程序的缺失率是个总结果,并没有告诉我们各个程序在其中的作用。需要更复杂的模型计算出共享缓存对单个程序的影响。

2.4 局部性的分解模型

传统的组合和分解模型使用重用距离和足迹。最早是文献[9,10]在分时系统(分时缓存共享)和文献[11]在多核系统(持续缓存共享)中提出的。在分时系统下,足迹测量只需要针对单一窗口长度。在多核系统下,足迹测量需要包括多个窗口长度。在之前的工作中,足迹是估算而不是实测的。

足迹理论给出两个更高效的组合分解模型:

(1)足迹转换模型:类似于传统模型,但是用足迹计算重用距离。传统模型的速度受限于重用距离的测量效率。虽然重用距离的测量得到很好的优化,但它仍然具有超线性的复杂度[12~15]。足迹转换模型的好处是它不再需要测量重用距离,建模速度可以加快几百倍。

(2)压力和灵敏度模型:程序的行为特征由压力和灵敏度两个函数来完全刻画。两个函数可以作为可视化的两条曲线。组合分解的过程很简单,只要查看两条曲线上的相关值。压力和灵敏度模型和足迹转换模型一样高效,但更直观、更容易使用。

2.5 足迹理论的应用

足迹模型的以下特点使其具有通用性和有效性:

(1)不依赖于硬件。足迹分析是基于数据访问而不是基于缓存访问的,它只需要一次足迹分析就可以推出所有缓存大小下的缺失率,并且结果不受分析方法和环境的影响。相比较而言,实机直接测量缺失率会不可避免地受到测量环境的影响。

(2)不依赖于测量环境。一个程序的足迹可以在共享环境中测量,不受其他程序干扰。其作用好比洁净室。相比较而言,直接测量受到先有鸡还是先有蛋的问题的困扰:一个程序的行为取决于它的同行,但同行的行为又取决于它本身。

(3)不依赖于同伴信息。一个程序的局部性度量完全决定于程序本身,不需要知道同伴程序。缓存共享的分析并不需要实际的缓存共享。共同运行的效果是靠计算,不是靠测量得到。

(4)组合而非测试。P 个程序有 $P(P-1)/2$ 个不同并行共享组合。足迹模型可以通过 P 个单程序运行测试,预测 $P(P-1)/2$ 并行运行的干扰。对于 P 个分析运行,可以逐个运行或几个并行提高速度。这种组合是静态的,不需要任何并行运行和测试。

使用组合分解的模型,我们能够回答一些长期悬而未决的问题,包括:

(1)是否有不依赖特定硬件的方式来比较程序对共享缓存的需求?不同应用领域的程序有何不同?

(2)共享缓存干扰和程序缺失率的关系是什么?较高的缺失率是否总是带来更大的干扰?

(3)缓存可以被看作是一个区域邦联,每个程序占据共享缓存的一部分。鉴于该分区是动态的、需求驱动的,动态空间划分是否比静态划分有固有的优势?

3 结束语

在多核环境下,程序在共享缓存中的相互作用日趋重要。局部性分析的近期发展产生了新的足迹理论。本文对足迹理论做了概括性介绍,包括足迹测量、度量转换和共享缓存中并行程序的效应组合和性能影响分解。足迹理论对共享缓存中程序相互干扰的现象提供了量化分析的高效、准确和直观的方法,以此可以提高程序在共享环境下的局部性认识和理解,增进多核系统上程序互动的管理和优化。

本文初稿是丁晨在 2013 年桂林召开的 CCF 高级系列讲座和中国高性能计算会议报告的中文讲稿的介绍部分。我们感谢组织召集者冯晓兵和张云泉的邀请和安排。英文稿是向晓娅博士论文的第一章。足迹理论基于她的博士工作,包括 2011 年 PACT 会议发表的计算平均足迹的线性算法(向氏公式)和 2013 年 ASPLOS 会议发表的高阶局部性理论,我们同时感谢罗切斯特大学白童心、鲍斌、罗昊,北京大学罗英伟、汪小林、李晔晨,密执根理工大学王振林和其他学者同行的合作,以

及《计算机工程与科学》杂志编辑的整理和排版。

参考文献:

[1] Zhang X, Dwarkadas S, Shen K. Towards practical page coloring-based multicore cache management[C]// Proc of the EuroSys Conference, 2009:89-102.

[2] Denning P J. Working sets past and present[J]. IEEE Transactions on Software Engineering, 1980,SE-6(1):68-84.

[3] Denning P J. The working set model for program behaviour [J]. Communications of ACM, 1968,11(5):323-333.

[4] Brock J, Luo H, Ding C. Locality analysis: A nonillion time window problem[C]//Proc of Big Data Analytics Workshop, 2013:1.

[5] Xiang X, Bao B, Bai T, et al. All-window profiling and composable models of cache sharing[C]//Proc of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2011:91-102.

[6] Xiang X, Bao B, Ding C, et al. Linear-time modeling of program working set in shared cache[C]//Proc of PACT, 2011: 350-360.

[7] Xiang Xiao-ya, Ding Chen, Luo Hao, et al. HOTL: A higher order theory of locality[C]//Proc of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, 2013:343-356.

[8] Wu M-J, Zhao M, Yeung D. Studying multicore processor scaling via reuse distance analysis[C]//Proc of ISCA, 2013, 499-510.

[9] Thiebaut D, Stone H S. Footprints in the cache[J]. ACM Transactions on Computer Systems, 1987,5(4):305-329.

[10] Suh G E, Devadas S, Rudolph L. Analytical cache models with applications to cache partitioning[C]// Proc of ICS, 2001:1-12.

[11] Chandra D, Guo F, Kim S, et al. Predicting inter-thread cache con tention on a chip multi-processor architecture[C] //Proc of HPCA, 2005:340-351.

[12] Schuff D L, Kulkarni M, Pai V S. Accelerating multicore reuse distance analysis with sampling and parallelization[C] //Proc of PACT, 2010:53- 64.

[13] Cui H, Yi Q, Xue J, et al. A highly parallel reuse distance analysis algorithm on GPUs[C]// Proc of IPDPS, 2012: 1080-1092.

[14] Niu Q, Dinan J, Lu Q, et al. PARDA: A fast parallel reuse distance analysis algorithm[C]//Proc of IPDPS, 2012: 1284-1294.

[15] Gupta S, Xiang P, Yang Y, et al. Locality principle revisited: A probability-based quantitative approach[C]//Proc of IPDPS, 2012:995-1009.

作者简介:

丁晨(1970 -),男,北京人,博士,教授,CCF 会员(E200016379M),研究方向为软件分析优化。**E-mail**:cding@cs. rochester. edu

DING Chen, born in 1970, PhD, professor,CCF member(E200016379M),his research interests include program analysis and optimization.

袁良(1984 -),男,河北保定人,博士,助理研究员,CCF 会员(E200013671M),研究方向为并行计算模型。**E-mail**:yuanliang@ict. ac. cn

YUAN Liang, born in 1984, PhD, assistant researcher, CCF member(E200013671M), his research interest includes parallel computational model.

Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance

Song Jiang and Xiaodong Zhang, *Senior Member*, *IEEE*

Abstract—Although the LRU replacement algorithm has been widely used in buffer cache management, it is well-known for its inability to cope with access patterns with weak locality. Previously proposed algorithms to improve LRU greatly increase complexity and/or cannot provide consistently improved performance. Some of the algorithms only address LRU problems on certain specific and predefined cases. Motivated by the limitations of existing algorithms, we propose a general and efficient replacement algorithm, called *Low Inter-reference Recency Set* (LIRS). LIRS effectively addresses the limitations of LRU by using recency to evaluate Inter-Reference Recency (IRR) of accessed blocks for making a replacement decision. This is in contrast to what LRU does: directly using recency to predict the next reference time. Meanwhile, LIRS mostly retains the simple assumption adopted by LRU for predicting future block access behaviors. Conducting simulations with a variety of traces of different access patterns and with a wide range of cache sizes, we show that LIRS significantly outperforms LRU and outperforms other existing replacement algorithms in most cases. Furthermore, we show that the additional cost for implementing LIRS is trivial in comparison with that of LRU. We also show that the LIRS algorithm can be extended into a family of replacement algorithms, in which LRU is a special member.

Index Terms—Operating systems, memory management, replacement algorithms.

1 Introduction

1.1 The Problems of the LRU Replacement Algorithm

The effectiveness of cache block replacement algorithms is critical to the performance stability of I/O systems. The LRU (Least Recently Used) replacement is widely used in managing buffer cache due to its simplicity, but many anomalous behaviors have been found with some typical workloads, where the hit rates of LRU may only slightly increase with a significant increase of cache size. The observations reflect LRU's inability to cope with access patterns with weak locality such as file scanning, regular accesses over more blocks than the cache size, and accesses on blocks with distinct frequency. Here are some representative examples reported in the research literature to illustrate how poorly LRU behaves:

1. Under the LRU algorithm, a burst of references to infrequently used blocks, such as sequential scans through large files, may cause the replacement of frequently referenced blocks in cache. This is a common complaint in many commercial systems: Sequential scans can cause interactive response time to deteriorate noticeably [17]. An effective replacement algorithm would be able to prevent hot blocks from being evicted by cold blocks.
2. For a cyclic (loop-like) pattern of accesses to a file that is only slightly larger than the cache size, LRU always mistakenly evicts the blocks that will be accessed the soonest because these blocks have not been accessed for the longest time [22]. A wise replacement algorithm would maintain a hit rate proportional to the buffer cache size.
3. In an example of multiuser database application, each record is associated with a B-tree index [17]. For a given number of records, assume their index entries can be packed into 100 blocks and 10,000 blocks are needed to hold the records. We use $R(i)$ to represent an access to Record i and $I(i)$ to Index i. The database application alternates its references to random index blocks and to the record blocks in the access sequence of $I(1)$, $R(1)$, $I(2)$, $R(2)$, $I(3)$, $R(3)$, ... Thus, the index blocks will be referenced with a probability of 0.005 and the data blocks are with a probability of 0.00005. Suppose that the cache can only hold 101 blocks. Ideally, all 100 index blocks are cached and only one record block is cached. However, LRU caches the 101 most recently accessed blocks. So, LRU keeps an equal number of index and record blocks in the cache and perhaps even more record blocks than index blocks. An intelligent replacement algorithm would choose the resident blocks according to their reference probability. Only those blocks with relatively high access probability deserve to stay in the cache for a longer time.

The reason for LRU to behave poorly in these situations is that LRU makes a bold assumption—a block that has not

- S. Jiang is with the Performance and Architecture (PAL) Group, Los Alamos National Laboratory, CCS-3, B256, PO Box 1663, Los Alamos, NM 87545. E-mail: sjiang@lanl.gov.
- X. Zhang is with the Computer Science Department, College of William and Mary, Williamsburg, VA 23187. E-mail: zhang@cs.wm.edu.

Manuscript received 26 Nov. 2003; revised 5 Nov. 2004; accepted 2 Mar. 2005; published online 15 June 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0227-1103.

0018-9340/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

been accessed for the longest time would wait for the longest time to be accessed again. This assumption cannot capture the access patterns exhibited in those workloads with weak locality. Generally speaking, there is less locality in buffer caches than that in CPU caches or virtual memory systems [20].

Meanwhile, LRU has its distinctive merits: simplicity and adaptability. It only samples and makes use of very limited history information—recency. While addressing the weakness of LRU, existing algorithms either take more history information into consideration, such as LFU (Least Frequently Used)-like ones in the cost of simplicity and adaptability or switch temporarily from LRU to other algorithms whenever certain predefined regularities are detected. In the switch-based approach, these algorithms actually act as supplements of LRU in a case-by-case fashion. To make a prediction of future access times, these algorithms assume the existence of a relationship between the future reference of a block with the behaviors of those blocks in its temporal or spatial locality, while LRU only associates the future behavior of a block with the block's own previous reference. This additional assumption increases the complexity of their implementations as well as their performance dependence on some specific characteristics of workloads. The replacement algorithm we propose, called LIRS, only samples and makes use of the same history information as LRU does—recency, and mostly retains the LRU assumption. Thus, it is simple and adaptive. In our design, LIRS does not directly target specific LRU problems, but fundamentally addresses the limitations of LRU.

1.2 An Executive Summary of Our Algorithm

We use recent Inter-Reference Recency (IRR) as the history information of a block, where the IRR of a block refers to the number of other distinct blocks accessed between two consecutive references to the block (IRR is also called reuse distance in some literature). In contrast, recency refers to the number of other distinct blocks accessed from last reference to the current time. We refer to the IRR between the last and the second-to-last references to a block as recent IRR or simply call it IRR without ambiguity in the rest of the paper. We assume that if the IRR of a block is large, the next IRR of the block is likely to be large. Following this assumption, we select the blocks with large IRRs for replacement because it is highly possible that these blocks will be evicted later by LRU before being referenced again under our assumption. It is noted that these evicted blocks may have been recently accessed, i.e., each has a small recency.

By adequately considering IRR in history information in our algorithm, we are able to eliminate negative effects caused by only considering recency, such as the problem shown in the aforementioned examples. When deciding which block to evict, our algorithm utilizes the block IRR information. It dynamically and responsively distinguishes low IRR (denoted as LIR) blocks from high IRR (denoted as HIR) blocks and keeps the LIR blocks in the cache, where the block recency is only used to help determine the LIR or HIR statuses of the blocks. We maintain an LIR block set and an HIR block set and manage to limit the size of the LIR set so that all the LIR blocks fit in the cache. The blocks in the LIR set are not selected for replacement and there are no misses for the references to these blocks. Only a very small portion of cache is allocated to store HIR blocks. Resident HIR blocks may be evicted at any recency. However, when the recency of an LIR block increases to a certain value and an HIR block gets accessed at a smaller recency than that of the LIR block, the statuses of the two blocks are switched. We name the proposed algorithm *Low Inter-reference Recency Set* (denoted as LIRS) replacement because the LIR set is what the algorithm tries to identify and keep in the cache. LIRS aims at addressing three issues in designing replacement algorithms: 1) how to effectively utilize multiple sources of history access information, 2) how to dynamically and responsively distinguish blocks by comparing their possibility to be referenced in the near future, and 3) how to minimize implementation overheads.

In the next section, we give an overview of the related work and highlight our technical contributions. The LIRS algorithm is described in Section 3. In Section 4, we present the trace-driven simulation results for performance evaluation and comparisons. We provide sensitivity and overhead analysis of the proposed replacement algorithm in Section 5 and conclude the paper in Section 6.

2 Related Work

The LRU replacement is widely used for the management of virtual memory, file buffer caches, and data buffers in database systems. The three representative problems described in the previous section are found in the different application fields. Many efforts have been made to address the LRU problems. We classify existing algorithms into three categories: 1) replacement algorithms based on user-level hints, 2) replacement algorithms based on tracing and utilizing history information of block accesses, and 3) replacement algorithms based on regularity detections.

2.1 User-Level Hints

Application-controlled file caching [3] and application-informed prefetching and caching [19] are the schemes based on user-level hints. These schemes identify blocks less likely to be reaccessed in the near future based on the hints provided by user programs. To provide appropriate hints, programmers need to understand the data access patterns, which adds to the programming burden. In [15], Mowry et al. attempted to abstract hints by compilers to facilitate I/O prefetching. In contrast, the LIRS algorithm can adapt its behavior to different access patterns without explicit hints. While the hint-based methods are orthogonal to the LIRS replacement, the collected hints may help LIRS refine the correlation of consecutive IRRs.

2.2 Tracing and Utilizing History Information

Realizing that LRU only utilizes limited access information, some researchers have proposed several algorithms to collect and use "deeper" history information, which include the LFU-like algorithms such as FBR, MQ, LRFU, as well as LRU-K and 2Q. We adopt a similar approach by effectively collecting and utilizing access information to design the LIRS replacement.

Robinson and Devarakonda proposed a frequency-based replacement algorithm (FBR) by maintaining reference counts for the purpose to "factor out" locality [20]. Zhou et al. proposed Multi-Queue (MQ), which sets up multiple queues and uses access frequencies to determine which

queue a block should be in [23]. However, it is slow for the frequency-based algorithms to respond to reference frequency changes and some of their parameters have to be found by trial and error. Having analyzed the advantages and disadvantages of LRU and LFU, Lee et al. proposed LRFU by combining them through weighing block recency and frequency factors [14]. The performance of the LRFU algorithm largely relies on a parameter called λ, which determines the relative weight of LRU or LFU and has to be adjusted according to the system configurations, even according to different workloads. However, LIRS does not have a tunable parameter that is sensitive to workloads.

The LRU-K algorithm addresses the LRU problems presented in examples 1 and 3 in the previous section [17]. LRU-K makes its replacement decision by comparing the times of the Kth-to-last references to blocks. After such a comparison, the oldest resident block is evicted. For simplicity, the authors recommended K = 2. By taking the time of the second-to-last reference to a block as the basis for comparison, LRU-2 can quickly remove cold blocks from the cache. However, for blocks without significant differences of reference frequencies, LRU-2 does not work well. In addition, LRU-2 is expensive: Each block access requires $\log(N)$ operations to manipulate a priority queue, where N is the number of blocks in the cache.

Johnson and Shasha proposed the 2Q algorithm that has constant time overhead [10]. They showed that the algorithm performs as well as LRU-2. The 2Q algorithm can quickly remove sequentially referenced blocks and loopingly referenced blocks with long looping intervals out of the cache. This is achieved by using a special buffer, called queue *A1in*, in which all missed blocks are initially placed. When the blocks are replaced from the *A1in* queue in a FIFO order, the addresses of those replaced blocks are temporarily placed in a ghost buffer called queue *A1out*. When a block is rereferenced, it is promoted to a main buffer called queue *Am* if its address is in the *A1out* queue. That is, only blocks that have a short reuse distance measured in *A1in* and *A1out* can be cached for a long time in *Am*. In this way, they are able to distinguish frequently referenced blocks from those infrequently referenced. By setting the sizes of the *A1in* and *A1out* queues as constants Kin and $Kout$, respectively, 2Q provides a victim block either from *A1in* or from *Am*. However, Kin and $Kout$ are predetermined parameters, which need to be carefully tuned and are sensitive to the types of workloads. While both 2Q and LIRS have simple implementations with low overheads, LIRS has overcome the drawbacks of 2Q by properly updating the LIR block set. Another recent algorithm, ARC, maintains two variable-size lists [16]. Their combined size is two times the number of blocks that are held in the cache. One half of the lists contain the blocks in the cache and the other half are for the history access information of replaced blocks. The first list contains the blocks that have been seen only once recently (cold blocks) and the second list contains the blocks that have been seen at least twice recently (hot blocks). The buffer spaces allocated to the blocks in these two lists are adaptively changed, depending upon in which list recent misses take place. More buffer spaces will serve cold blocks (respectively, hot blocks) if there are more cold block (respectively, hot block) accesses. However, although the authors advocated the superiority of the ARC algorithm with its adaptiveness and avoidance of tunable parameters, the locality of the blocks in the two lists, quantified by recency or frequency, cannot be directly and consistently compared. For example, a block that is regularly accessed with an IRR a little bit more than the cache size may have no hits at all, while a block in the second list can stay in the cache without any accesses since it has been accepted into the list.

The Inter-Reference Gap (IRG) of a block is the number of the references between consecutive references to the block, which is different from IRR on whether duplicate references to a block are counted. Phalke and Gopinath considered the correlation between history IRGs and future IRGs [18]. The past string of IRGs of a block is modeled by Markov chain to predict its next IRG. However, as Smaragdakis et al. indicated, replacement algorithms based on a Markov model fail in practice because they try to solve a much harder problem than the replacement problem itself [22]. An apparent difference in their algorithm from the LIRS algorithm is in how to measure the distance between two consecutive references to a block. Our study shows that IRR is more justifiable than IRG in this circumstance. First, IRR only counts the distinct blocks and filters out high-frequency events, which may be volatile with time. Thus, the IRR is more relevant to the next IRR than the IRG to the next IRG. Moreover, it is the "recency" rather than the "gap" information that is used by LRU. An elaborate argument favoring IRR in the context of virtual memory page replacement can be found in [22]. Second, IRR can be easily dealt with under the LRU stack model [2], on which most popular replacements are based.

2.3 Detection and Adaptation of Access Regularities

More recently, some researchers took another approach to detect access regularities from the history information by relating the accessing behavior of a block to those of the blocks in its temporal or spatial locality scope. Then, different replacements, such as Most Recently Used (MRU), can be applied to those blocks with specific access regularities.

Glass and Cao proposed the SEQ algorithm for adaptive page replacement in virtual memory management [9]. It detects sequential address reference patterns. If a long sequence of page faults with continuous addresses is found, MRU is applied to the sequence. If such a sequence is not detected, SEQ performs the LRU replacement. These detections only take place when there are page faults, so it has a low overhead acceptable in virtual memory management. However, Smaragdakis et al. argued that address-based detection lacks generality and advocated using aggregate recency information to characterize page behaviors [22]. Their EELRU examines aggregate recency distributions of accessed pages and changes the page eviction points using an online cost/benefit analysis by assuming the correlation among tempo-rally contiguously referenced pages. This is different from LRU, which actually always sets the eviction point at the bottom of the LRU stack. However, EELRU has to choose an eviction point from a predetermined set of LRU stack positions. And, the way to select the set affects its performance. Moreover, by an aggregate analysis, EELRU cannot quickly respond to the changing access patterns. Without spatial or temporal detections, LIRS uses the independent recency events of each block to effectively characterize their references.

Choi et al. proposed an adaptive buffer management algorithm called DEAR, which automatically detects the block reference patterns of applications and applies different replacement algorithms to different applications based on their detected reference patterns [5]. Further, they proposed an Application/File-level Characterization (AFC) algorithm in [4], which first detects the reference characteristics at the application level and then at the file level if necessary. Accordingly, appropriate replacement algorithms are used to the blocks with different patterns. The Unified Buffer Management (UBM) algorithm by Kim et al. also detects patterns in the recorded history [13]. Unlike the detection method used in DEAR, which associates the backward distance and frequency with the forward distances of blocks between two consecutive detection invocation points, UBM tracks the reference information such as the file descriptor, start block number, end block number, and loop period if a rereference occurs. More recently, Gniady et al. proposed the PCC replacement algorithm, which conducts its access pattern detection on a per-system-call-site basis to improve the detection accuracy and efficiency [8]. Although these elaborate detections of access patterns provide a large potential for significant performance improvements, they addressed the LRU problems in a case-by-case fashion and have to deal with the allocation problem, which does not appear in LRU. To facilitate the online evaluation of buffer utilizations, certain premeasurements are needed to set predefined parameters used in the buffer allocation schemes [4], [5], [8]. LIRS does not have these design challenges. While it chooses the victim block in a global stack as LRU does, it can take the advantages provided by the detection-based algorithms.

More work on program locality analysis, prediction, and enhancement is conducted in the program behavior studies using static compiler analysis, data profiling, and runtime data analysis techniques (e.g., see [6]). There are two major differences between these studies and those on replacement algorithms in operating systems. First, program behavior studies are usually conducted at a finer level such as data elements and instructions rather than at the block or page level defined by the system. Usually, they require much more computing effort, which could be too expensive for a replacement algorithm running in the operating system. Second, program behavior studies focus on understanding the behavior of a specific program. It doesn't consider system parameters such as memory size and interaction among simultaneously running programs. However, a replacement algorithm must be designed from the system perspective, taking both the properties of workloads and system configurations into consideration. These constraints prevent the replacement algorithm from conducting an aggressive locality analysis or pattern detection. Thus, a simple yet effective replacement algorithm becomes a critical system design issue.

3 THE LIRS ALGORITHM

3.1 General Idea

We classify referenced blocks into two sets: High Inter-reference Recency (HIR) block set and Low Inter-reference Recency (LIR) block set. Each block with its history information in cache has a status—either LIR or HIR. Some HIR blocks may not reside in the cache, but keep their metadata in the cache, recording their status as nonresident HIR. We divide the cache, whose size in blocks is L, into a major part and a minor part in terms of their sizes. The major part, with its size of L_{lirs}, is used to store LIR blocks and the minor part, with its size of L_{hirs}, is used to store blocks from HIR block set, where $L_{lirs} + L_{hirs} = L$. When a miss occurs and a block is needed for replacement, we choose an HIR block that is resident in the cache. The blocks in the LIR block set always reside in the cache, i.e., there are no misses for the references to the LIR blocks. However, a reference to an HIR block is likely to encounter a miss because L_{hirs} is very small (its practical size can be as small as 1 percent of the cache size).

We use Table 1 as a simple example to illustrate how a replaced block is selected by the LIRS algorithm and how LIR/HIR statuses are maintained. In Table 1, symbol "X" denotes a block access at a virtual time.[1] As an example, block A is accessed at times 1, 6, and 8. Based on the definition of recency and IRR in Section 1.2, at time 10, blocks A, B, C, D, E have their IRR values of 1, 1, "infinite," 3, and "infinite," respectively, and have their recency values of 1, 3, 4, 2, and 0, respectively. We assume the cache can hold three blocks, $L_{lirs} = 2$ and $L_{hirs} = 1$, thus, at time 10, the LIRS algorithm leaves two blocks in the LIR set (the LIR set = {A, B}). The rest of the blocks go to the HIR set (the HIR set = {C, D, E}). Because block E is the most recently referenced, it is the only resident HIR block due to $L_{hirs} = 1$. If there is a reference to an LIR block, we keep it in the LIR block set. If there is a reference to an HIR block, we need to know whether we should change its status to LIR.

The key to successfully making the LIRS idea work in practice rests on whether we are able to dynamically and responsively maintain the LIR block set and HIR block set. When an HIR block is referenced, it gets a new IRR equal to its recency. Then, we determine whether the new IRR should be considered small relative to the current LIR blocks so that we know whether we need to change its status to LIR. Here, we have two options: compare the new IRR either with the IRRs or with the recencies of the LIR blocks. We take the recencies for the comparison for two reasons: 1) The IRRs are generated before their respective recencies and may be outdated, which is not directly relevant to the new IRR of the HIR block. A recency of a block is determined not only by its own reference activity, but also by the recent activities of other blocks. The outcome of comparing the new IRR and the recencies of the LIR blocks determines the eligibility of the HIR block to be considered as a hot block. While we state that IRRs are used to determine which blocks should be replaced, it is the new IRRs that are directly used in the comparisons. 2) If the new IRR of the HIR block is smaller than the recency of an LIR block, it will be smaller than the upcoming IRR of the LIR block. This is because the recency of the LIR block is a part of its upcoming IRR and not greater than the IRR. Thus, the comparisons with the recencies are actually the comparisons with the relevant IRRs. Once we know that the new IRR of the HIR block is smaller than the maximum recency of all the LIR blocks, we switch the LIR/HIR statuses of the HIR block and the LIR block with the maximum recency. Following this rule, we can 1) allow an HIR block with a relatively small IRR to join the LIR block

1. Virtual time is defined on the reference sequence, where a reference represents a time unit.

TABLE 1
An Example to Explain How a Victim Block Is Selected by the LIRS Algorithm and How LIR/HIR Statuses Are Maintained

Blocks / Virtual time	1	2	3	4	5	6	7	8	9	10	Recency	IRR
A	x					x		x			1	1
B			x		x						3	1
C				x							4	inf
D		x					x				2	3
E									x		0	inf

An "X" refers to the block in a row that is referenced at the virtual time of a column. The recency and IRR columns represent their respective values at virtual time 10 for each block. We assume $L_{lirs} = 2$ and $L_{hirs} = 1$ and, at time 10, the LIRS algorithm leaves two blocks in the LIR set (= {A, B}) and the HIR set is {C, D, E}. The only resident HIR block is E.

set in a timely fashion by replacing a LIR block from the set and 2) keep the size of LIR block set no larger than L_{lirs}, thus the entire set of blocks can reside in the cache.

Again, in the example of Table 1, if there is a reference to block D at time 10, a miss occurs. The LIRS algorithm replaces resident HIR block E, instead of block B, which would be replaced by LRU due to its largest recency. Furthermore, because block D is referenced, its new IRR is 2, which is smaller than the recency of LIR block B (= 3), indicating that the upcoming IRR of block B will not be smaller than 3. So, the status of block D is switched to LIR and the block joins the LIR block set, while block B becomes an HIR block. Since block B becomes the only resident HIR block, it is going to be evicted from the cache once another free block is requested. If, at virtual time 10, block C, with its recency of 4, rather than block D, with its recency of 2, gets accessed, there will be no status switching. Then, block C becomes a resident HIR block, while the replaced block is still E at virtual time 10. In this way, the LIR block set and HIR block set are formed and dynamically maintained.

3.2 The LIRS Algorithm Based on LRU Stack

The LIRS algorithm can be efficiently built on the model of LRU stack, which is an implementation structure of LRU. The LRU stack contains L entries, each of which represents a block.[2] Usually, L is the cache size in blocks. The LIRS algorithm makes use of the stack to keep track of recency and to dynamically maintain LIR block set and HIR block set. In contrast to the LRU stack, where only resident blocks are managed by the LRU algorithm in the stack, we store LIR blocks and HIR blocks with their recencies less than the maximum recency of the LIR blocks in a stack called LIRS stack S. S is similar to the LRU stack in operation but has a variable size. With this design, we do not need to explicitly record the IRR and recency values and to search for the maximum recency value. Each entry in the stack records the LIR/HIR status of a block and its residence status, indicating whether or not the block resides in the cache. To facilitate the search of the resident HIR blocks, we link all these blocks into a small stack, Q, with its size of L_{hirs}. Once a free block is needed, the LIRS algorithm removes a resident HIR block from the bottom of stack Q for replacement. However, the replaced HIR block remains in stack S with its residence status changed to "nonresident" if it is originally in the stack. We ensure the block in the bottom of stack S is an LIR block by removing HIR blocks below it. Once an HIR block in the LIRS stack gets referenced, which means there is at least one LIR block whose upcoming IRR will be greater than the new IRR of the HIR block (such as the one at the bottom of the stack), we switch the LIR/HIR statuses of the HIR block and the LIR block at the bottom. Then, the LIR block at the bottom is evicted from stack S and goes to the top of stack Q as a resident HIR block. This block will soon be replaced from the cache due to the small size of stack Q (at most L_{hirs}).

Such a design is partially inspired by the observation of improper LRU replacement behavior: If a block is evicted from the bottom of an LRU stack, it means the block occupies a buffer during the period of time when it moves from the top to the bottom of the stack without being referenced. Why do we have to afford a buffer for another long idle period when the block is loaded into the cache the next time as what LRU does? The rationale for the correction of the LRU decision is the assumption that temporal IRR locality holds for block references.

3.3 A Detailed Description

In the LIRS replacement, there is an operation called "stack pruning" on LIRS stack S, which removes the HIR blocks at the stack bottom until an LIR block sits there. This operation serves two purposes: 1) We ensure the block at the stack bottom always belongs to the LIR block set. 2) After the LIR block in the bottom is removed, those HIR blocks contiguously located above it will not have a chance to change their status from HIR to LIR since their recencies are larger than the new maximum recency of the LIR blocks.

When an LIR block set is not full, all the accessed blocks are given LIR status until its size reaches L_{lirs}. After that, HIR status is given to any blocks that are accessed for the first time and to blocks that have not been accessed for a long time so that currently they are not in stack S.

Fig. 1 shows a scenario where stack S holds three types of blocks, LIR blocks, resident HIR blocks, nonresident HIR blocks, and stack Q holds all of the resident HIR blocks. An HIR block could either be in stack S or not. Fig. 1 does not depict the nonresident HIR blocks that are not in stack S. There are three cases for the references to these blocks in the LIRS algorithm, which are also illustrated in Fig. 2, using the example shown in Table 1.

2. For simplicity, in the rest of the paper we use "a block in the stack" instead of "the entry of a block in the stack" without ambiguity.

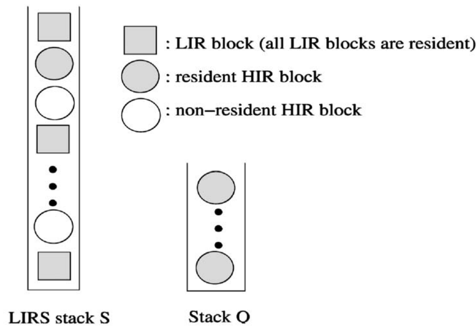


Fig. 1. LIRS stack S holds LIR blocks as well as some HIR blocks, with or without resident status, and stack Q holds all the resident HIR blocks.

1. **Upon accessing an LIR block X.** This access is guaranteed to be a hit in the cache. We move it to the top of stack S. If the LIR block is originally located at the bottom of the stack, we conduct a stack pruning. This case is illustrated in the transition from state (a) to state (b) in Fig. 2.
2. **Upon accessing an HIR resident block X.** This is a hit in the cache. We move it to the top of the stack S. There are two cases for the original location of block X: a) If X is in stack S, we change its status to LIR. This block is also removed from stack Q. The LIR block at the bottom of S is moved to the top of stack Q with its status changed to HIR. A stack pruning is then conducted. This case is illustrated in the transition from state (a) to state (c) in Fig. 2. b) If X is not in stack S, we leave its status unchanged and move it to the top of stack Q.
3. **Upon accessing an HIR nonresident block X.** This is a miss. We remove the HIR resident block at the bottom of stack Q (it then becomes a nonresident block) and evict it from the cache. Then, we load the requested block X into the freed buffer and place it at the top of stack S. There are two cases for the original location of block X: a) If X is in the stack S, we change its status to LIR and move the LIR block at the bottom of stack S to the top of stack Q with its status changed to HIR. A stack pruning is then conducted. This case is illustrated in the transition from state (a) to state (d) in Fig. 2. b) If X is not in stack S, we leave its status unchanged and place it at the top of stack Q. This case is illustrated in the transition from state (a) to state (e) in Fig. 2.

4 Performance Evaluation

4.1 Experiment Settings

We use trace-driven simulations with various types of workloads to evaluate the LIRS algorithm and compare it with other algorithms. We have adopted many application workload traces used in the previous studies aiming at addressing the LRU limitations. These are traces recording file access requests from one or multiple running applications, representing a wide range of access patterns, sizes, and sources. We have also generated a synthetic trace. Among these traces, *cpp*, *cs*, *glimpse*, and *postgres* are used in [4], [5] (*cs* is named as *cscope*, and *postgres* is named as *postgres*2 there), *sprite* is used in [14], *multi*1, *multi*2, and *multi*3 are used in [13]. We briefly describe the traces here.

1. **2-pools** is a synthetic trace which simulates application behavior described in the third example in Section 1.1. The trace contains 100,000 references.
2. **cpp** is a GNU C compiler preprocessor trace. The total size of C source programs used as input is roughly 11 MB.
3. **cs** is an interactive C source program examination tool trace. The total size of the C programs used as input is roughly 9 MB.
4. **glimpse** is a text information retrieval utility trace. The total size of the text files used as input is roughly 50 MB.
5. **postgres** is a trace of join queries among four relations in a relational database system from the University of California at Berkeley.
6. **sprite** is from the Sprite network file system, which contains requests to a file server from client workstations for a two-day period.
7. **multi1** is obtained by executing two workloads, *cs* and *cpp*, together.
8. **multi2** is obtained by executing three workloads, *cs*, *cpp*, and *postgres*, together.
9. **multi3** is obtained by executing four workloads, *cpp*, *gnuplot*, *glimpse*, and *postgres*, together. *gnuplot* is a popular graph plotting tool.

The only parameter of the LIRS algorithm, L_{hirs}, is set as 1 percent of the cache size or $L_{lirs} = 99\%$ of the cache size in the experiments. This selection results from a sensitivity study on the parameter, which is described in Section 5.1.

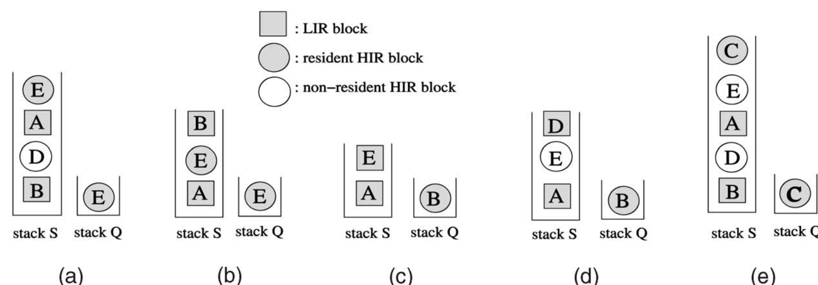


Fig. 2. Illustration of reference effects on the stacks using the example shown in Table 1. In the figure, (a) corresponds to the state at virtual time 9. References to B, E, D, or C at virtual time 10 result in states (b), (c), (d), and (e), respectively.

4.2 Access Pattern-Based Performance Evaluation

Through an elaborate investigation, Choi et al. classified the file cache access patterns into four types [4]:

- Sequential references: All blocks are accessed one after another and never reaccessed;
- Looping references: All blocks are accessed repeatedly with a regular interval (period);
- Temporally clustered references: Blocks accessed more recently are the ones more likely to be accessed in the near future;
- Probabilistic references: Each block has a stationary reference probability and all blocks are accessed independently with their associated probability.

The classification serves as a basis for their access pattern detections and for adapting to different replacement algorithms. For example, MRU applies to sequential and looping patterns, LRU applies to temporally clustered patterns, and LFU applies to probabilistic patterns. Though the LIRS algorithm does not rely on such a classification, we would like to use it to present and explain our experiment results. Because a sequential pattern is a special case of looping pattern (with infinite interval), we only use the last three types: looping, temporally clustered, and probabilistic patterns.

Algorithms LRU, LRU-2, 2Q, ARC, LRFU, and LIRS belong to the same replacement algorithm category. In other words, these algorithms take the same technical approach—predicting the access possibility of a block through its own history access information. Thus, we focus on the performance comparisons between LIRS and other algorithms in this category. As representative algorithms in the category of regularity detections, we choose two algorithms for comparisons: UBM for its spatial regularity detection and EELRU for its temporal regularity detection. UBM simulation requires the file ID, offset, and process ID of a reference. However, some traces available to us only consist of logical block numbers, which are unique numbers for the accessed blocks. Thus, we only produce the UBM simulation results for the traces used in paper [13], which are $multi1$, $multi2$, $multi3$. We also include the results of OPT, an optimum, offline replacement algorithm [2] for comparison.

We divide the traces into four groups based on their access patterns. Traces cs, $glimpse$, and $postgres$ belong to the looping type, traces cpp and $2\text{-}pools$ belong to the probabilistic type, trace $sprite$ belongs to the temporally clustered type, and traces $multi1$, $multi2$, and $multi3$ belong to the mixed type.

We present the performance results for each trace using a pair of figures: the time-space maps and the hit rate curves. In a time-space map, the x axis represents virtual time, a position in the reference sequence of a given workload, and the y axis represents the logical block numbers of the accessed blocks. The hit rate curves show the hit rates with different cache sizes for the various replacement algorithms on a workload trace.

4.2.1 Performance for Looping Type Workloads

Fig. 3 plots three pairs of time-space maps and the hit rate curves generated by the various algorithms for workloads cs, $glimpse$, and $postgres$, respectively. The time-space maps show that all three programs have looping patterns with long intervals. As expected, LRU performs poorly for these workloads with the lowest hit rates. Let us take cs as an example, which has a pure looping pattern. Each block is accessed at almost the same frequency. Since all blocks in a loop have the same eligibility to be kept in the cache, it is desirable to keep the same set of blocks in the cache no matter what blocks are referenced currently. That is indeed what LIRS does: The same set of LIR blocks is fixed in the cache because the HIR blocks do not have IRRs small enough to change their status. In the looping pattern, recency indicates the opposite of the future reference time of a block: The larger the recency of a block is, the sooner the block will be referenced. The hit rate of LRU for cs is almost 0 percent until the cache size approaches 1,400 blocks, which can hold all the accessed blocks in a loop. It is interesting to see that the hit rate curve of LRU-2 overlaps with the LRU curve. This is because LRU-2 selects the same victim block as the one selected by LRU for replacement. When making a decision, LRU-2 compares the second-to-last reference time, which is the recency plus the recent IRG. However, the IRGs are the same for all the blocks at any time after the first reference. Thus, LRU-2 relies only on recency to make its decision, the same as LRU does. In general, when recency makes a major contribution to the second-to-last reference time, LRU-2 behaves similarly to LRU.

Except for cs, the other two workloads have mixed looping patterns with various sizes of intervals. LRU exhibits the stair-step hit rate curves for the workloads. LRU is not effective until all the blocks in its locality scope are brought into the cache. For example, only after the cache can hold 355 blocks does the LRU hit rate curve of $postgres$ have a sharp increase from 16.3 percent to 48.5 percent. Because LRU-2 considers the last IRG in addition to the recency, it is easier for it to distinguish blocks with different loop intervals than LRU does. However, LRU-2 lacks the capability of dealing with the varying recencies of these blocks. Our experiments show that the performance improvement achieved by LRU-2 over LRU is limited.

It is illuminating to observe the performance difference between 2Q and LIRS because both employ two linear data structures following a similar principle that only rereferenced blocks deserve to be in cache for a longer time. We can see that the hit rates of 2Q are significantly lower than those of LIRS for all three workloads. As the cache size increases, 2Q even performs worse than LRU for workloads $glimpse$ and $postgres$. Another observation for 2Q on $glimpse$ and $postgres$ is a serious "Belady's anomaly" [1]: Increasing the cache size could reduce the number of hits. Although ARC is an adaptive algorithm without tunable parameters, it actually shares the same problem as 2Q. The performance improvement of ARC over LRU is very limited. Belady's anomaly also appears in $glimpse$ for ARC. This is mainly caused by the inconsistent quantification and comparison of block locality in the two lists of ARC. This issue has been effectively addressed in LIRS. We will provide an in-depth analysis on this issue in Section 4.3.

LRFU, which combines LRU and LFU, is not effective on workloads with a looping pattern because the block reference frequencies in looping references are hard to distinguish. As an example, the LRFU and LRU hit rate curves for workload cs are overlapped.

Our simulation results show LIRS significantly outperforms all of the other algorithms and its hit rate curves are very close to those of OPT. Meanwhile, the results also

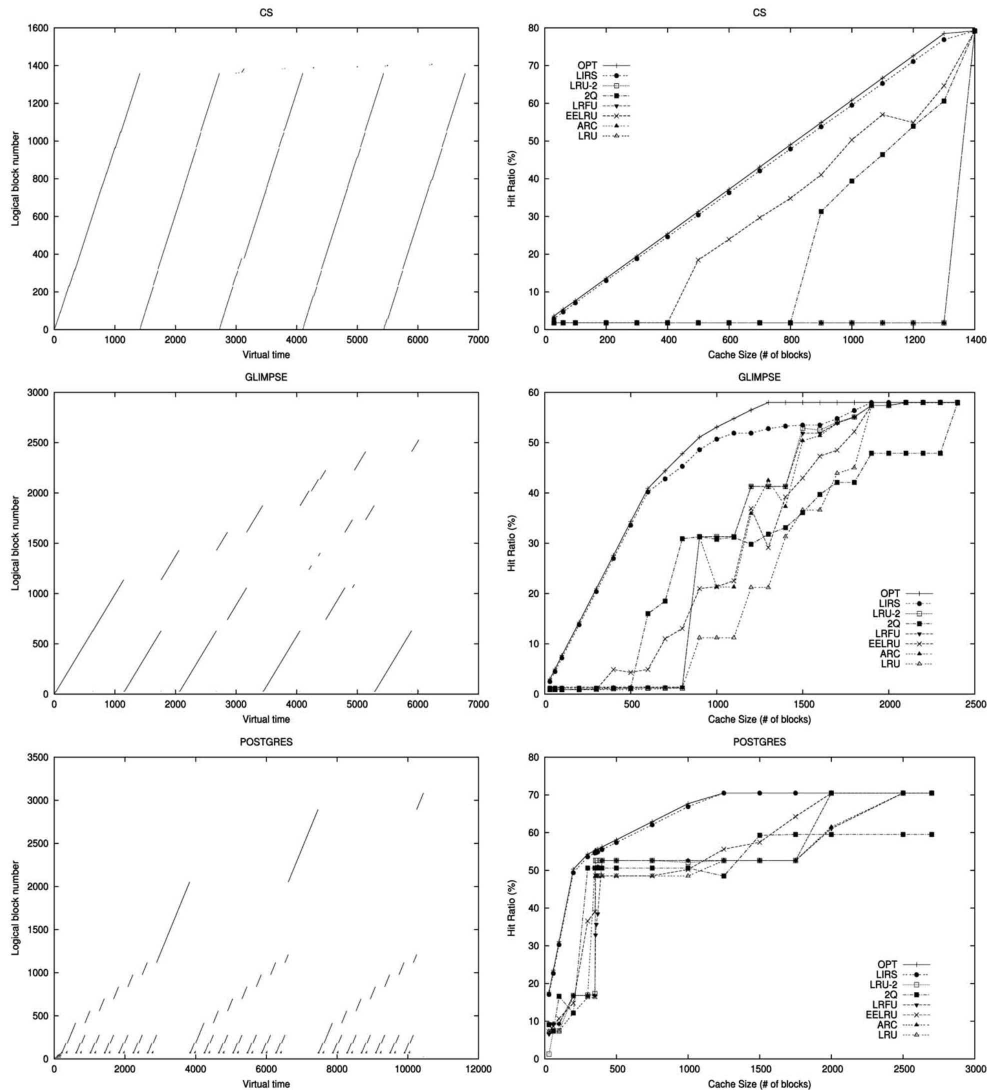


Fig. 3. The time-space maps and the hit rate curves of **cs**, **glimpse**, and **postgres** for the replacement algorithms.

show that the hit rates of *cs* and *postgres* are closer to those of OPT than the hit rates of *glimpse*. This indicates that LIRS can make a more accurate prediction on the future LIR/HIR statuses when the looping intervals are of less variance. Because *cs* and *postgres* have relatively fixed loop intervals, their consecutive IRRs are of less variance, which makes the IRR assumption hold well. However, the LIRS algorithm is not sensitive to the variance of IRRs, which is reflected by the significant hit rate improvements on workload *glimpse*. This is further evidenced by the results for the mixed pattern workloads described in Section 4.2.4.

4.2.2 Performance for the Probabilistic Type Workloads

Fig. 4 plots two pairs of time-space maps and the hit rate curves generated by the various replacement algorithms for traces *cpp* and *2-pools*, respectively. According to the detection results in [4], workload *cpp* exhibits a probabilistic reference pattern. In *cpp*, before the cache size increases to 100 blocks, the hit rates of LRU are much lower than those of LIRS. For example, when the cache size is 50 blocks, the hit rate of LRU is 9.3 percent, while the hit rate of LIRS is 55.0 percent. This is because holding a reference locality scope needs about 100 blocks. LRU cannot exploit the locality until enough cache space is available to hold all the recently referenced blocks. However, the capability for LIRS to exploit locality does not rely on the cache size—when it is identifying the LIR set, it always makes sure that the set will be able to fit in the cache. *2-pools* is generated to evaluate the replacement algorithms on their abilities to recognize the long-term reference behaviors. Though the reference frequencies are very different between the record blocks and the index blocks, it is hard for LRU to distinguish them when the cache size is small relative to the number of the referenced blocks because LRU takes only recency into consideration. The LRU-2, 2Q, and LIRS algorithms take one more previous reference into consideration—the time for the second-to-last reference to a block is involved. Even though the reference events to a block are randomized (i.e., the IRRs of a block are random with a certain fixed frequency, which is unfavorable to LIRS), LIRS still outperforms LRU-2 and 2Q. However, LRFU utilizes "deeper" history information. The constant long term frequency becomes more visible to the LFU-like algorithm. Thus, the performance of LRFU is slightly better than that of LIRS. It

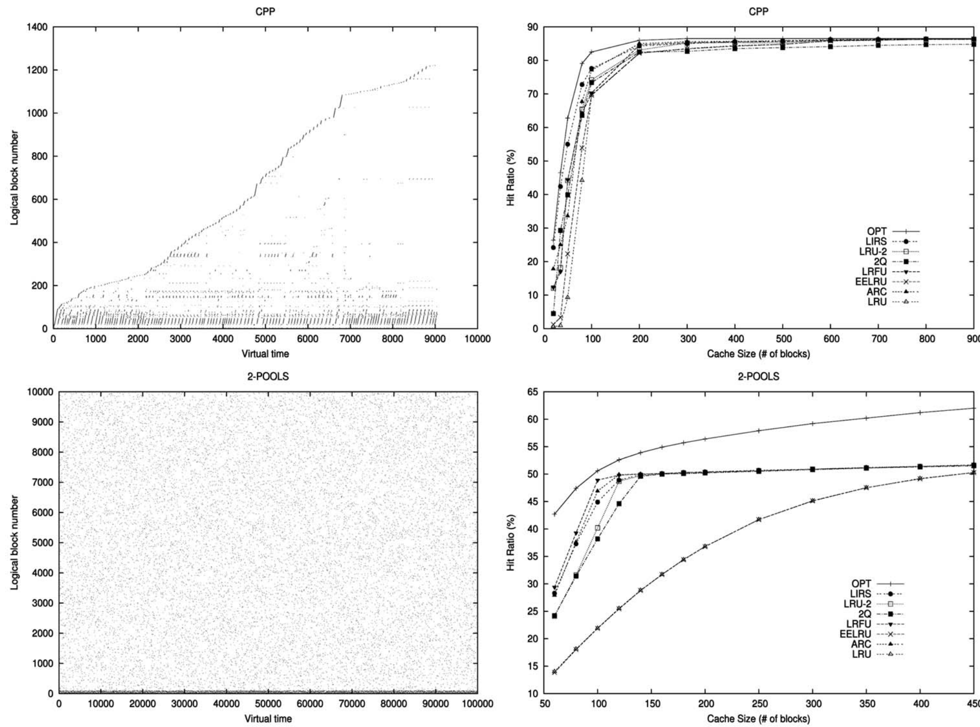


Fig. 4. The time-space maps and the hit rate curves of **cpp** and **2-pools** for the replacement algorithms.

is not surprising to see that the hit rate curve of EELRU overlaps with that of LRU, showing its poor performance. This is because EELRU relies on an analysis of a temporal recency distribution to decide whether to conduct an early point eviction. In *2-pools*, the blocks with high access frequency and the blocks with low access frequency are alternatively referenced, thus no sign of an early point eviction can be detected.

4.2.3 Performance for Temporally Clustered Type Workloads

Fig. 5 presents the time-space map of workload $sprite$ and its hit rate curves generated by the various replacement algorithms. $sprite$ exhibits a temporally clustered reference pattern. Fig. 5 shows that the LRU hit rate curve smoothly climbs with the increase of the cache size. Although there is still a gap between the LRU and OPT curves, the slope of the LRU curve is close to the OPT curve. $sprite$ is a so-called LRU-friendly workload [22], which seldom accesses more blocks than the cache size over a fairly long period of time. For this type of workload, the behavior of the other algorithms should be similar to that of LRU so that their hit rates could be close to those of LRU. Before the cache size reaches 350 blocks, the hit rates of LIRS are higher than those of LRU. After that point, the hit rates of LRU become slightly higher. Here is the reason for the slight performance degradation of LIRS beyond that cache size: Whenever there is a locality scope shift or transition, that is, some HIR blocks get referenced, one more miss than would occur in LRU may be experienced by an HIR block. Only the next reference to the block in the near future after the miss makes it switch from HIR to LIR status and then remain in the cache. However, because of the strong locality, there are not frequent locality scope changes. So, the negative effect of the extra misses is limited.

4.2.4 Performance for Mixed Type Workloads

Fig. 6 presents three pairs of time-space maps and the hit rate curves generated by the various replacement algorithms for workloads $multi1$, $multi2$, and $multi3$. The authors in [13] provided a detailed discussion why their UBM shows the best performance among the algorithms they have considered—UBM, LRU-2, 2Q, and EELRU. Here, we focus on performance difference between LIRS and UBM. UBM is a typical spatial regularity detection-based replacement algorithm that makes exhaustive reference pattern detections. UBM tries to identify sequential and looping patterns and applies MRU to the detected patterns. UBM further measures looping intervals and conducts period-based replacements. For those unidentified blocks without special patterns, LRU is applied. A scheme for dynamically allocating buffers among the blocks managed by different algorithms is employed. Without devoting specific efforts to specific regularities, LIRS outperforms UBM for all three mixed type workloads, which indicates that our assumption on IRR holds well and LIRS is able to cope with weak locality in the workloads with mixed type patterns.

4.3 LIRS versus Other Stack-Based Replacements

To get insights into the superiority of LIRS over other stack-based replacement algorithms, including LRU, 2Q, we plot a time-IRR graph to observe their actions on the blocks accessed at different recencies. In a time-IRR graph, the x axis represents virtual time, a reference in the access stream, the y axis represents IRR, the recency where the reference at a virtual time takes place. For first time accessed blocks, their IRRs are infinite, which we do not plot in the graph. We select two representative workloads, a non-LRU-friendly one, $postgres$, and an LRU-friendly one, $sprite$, for this study. Their IRRs are depicted in Fig. 7.

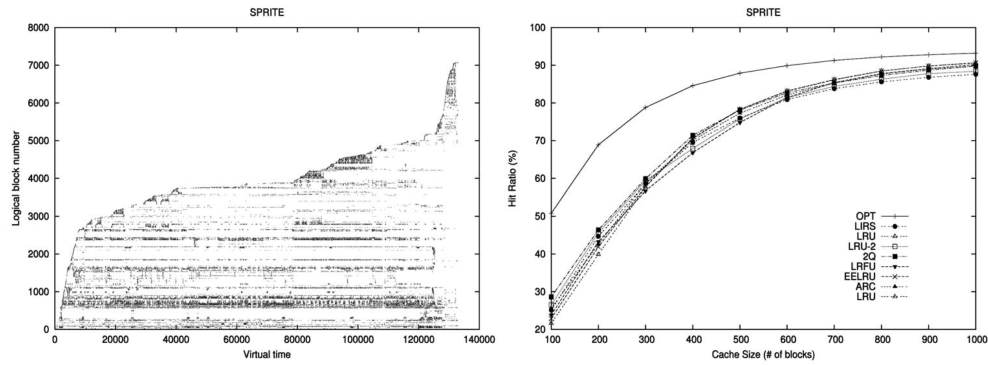


Fig. 5. The time-space map and the hit rate curve of **sprite** for the replacement algorithms.

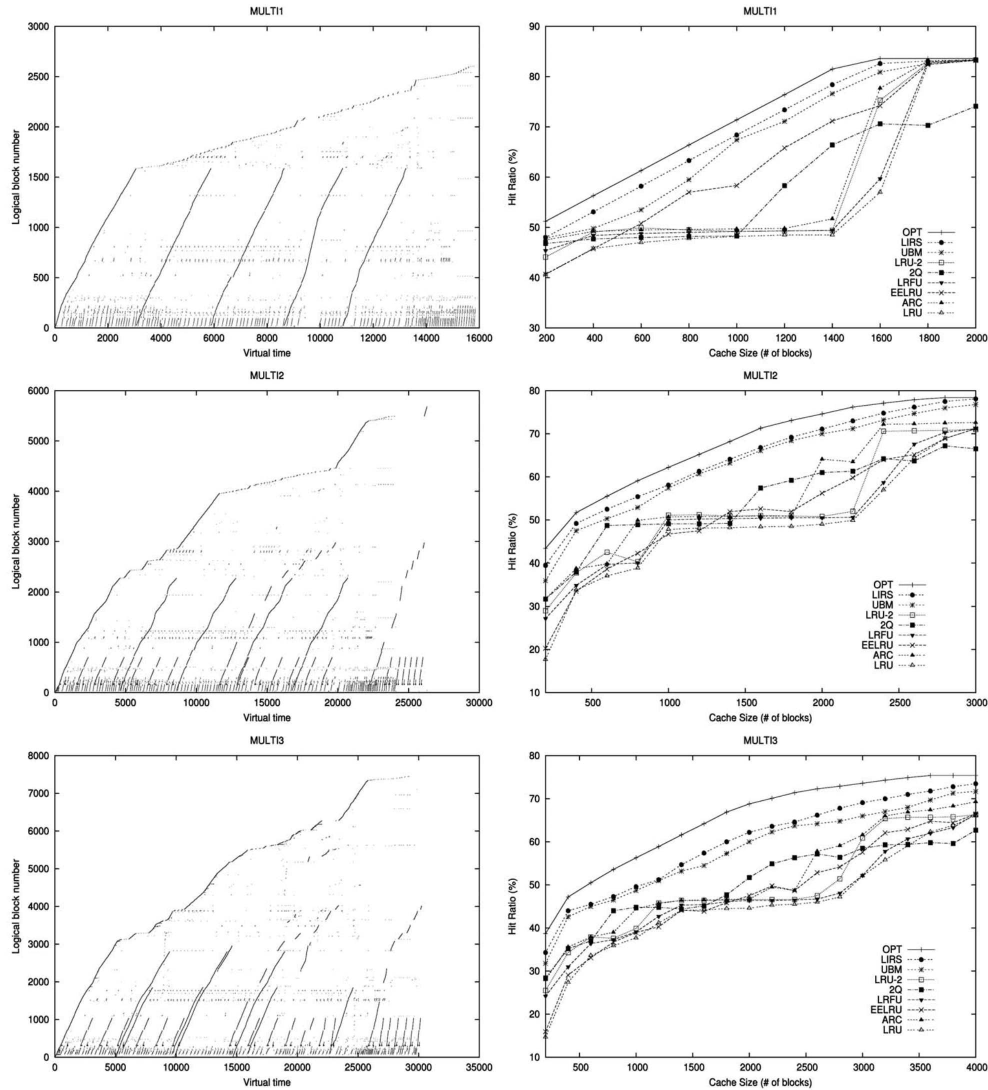


Fig. 6. The time-space maps and the hit rate curves of **multi1**, **multi2**, and **multi3** for the replacement algorithms.

The stack size of LRU, which is determined by the cache size in blocks, is fixed. If the stack size is L, all the references shown in the graphs with their IRRs less than L are hits and those with IRRs larger than L are misses in LRU. Thus, the hit rates of LRU are determined by the IRR distribution. If most of the IRRs are concentrated in the low recency area, such as what is shown in the graph for *sprite*, LRU will get a high hit rate. For workloads with dispersed recency distributions, LRU is incompetent in achieving high hit rates. For example, in *postgres*, there are two IRR concentrations at around IRRs 350, 1150, and 1950. In corresponding to the IRR distribution, there are some apparent "lift ups" in the LRU hit rate curve when the cache size reaches these values (see Fig. 3). If there are a large number of

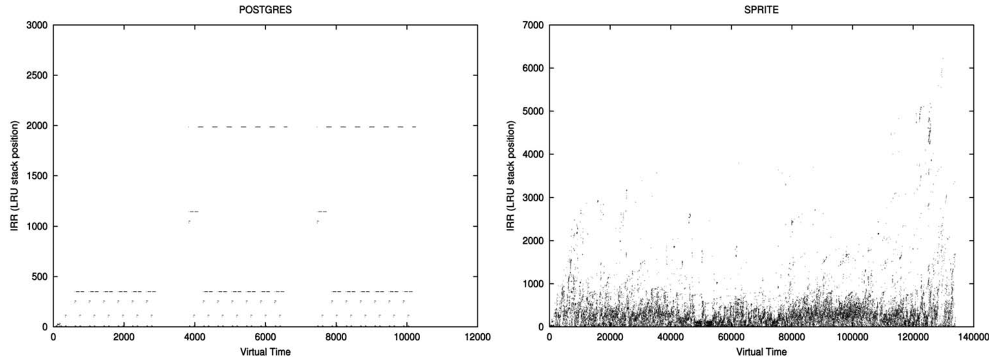


Fig. 7. The IRRs of the references in **postgres** and **sprite**.

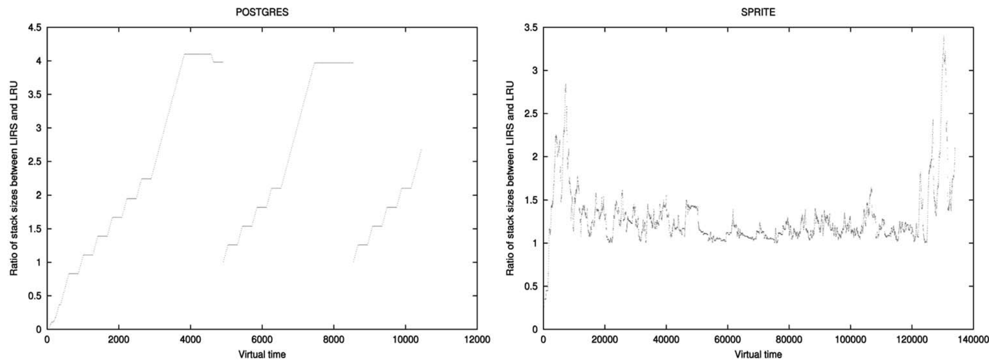


Fig. 8. The ratios of LIRS stack size and LRU stack size for **postgres** and **sprite**. Cache size is 500.

references with their IRRs larger than the LRU stack size, many blocks with their low recencies but high IRRs would hold the stack spaces (residing in the cache) without being accessed before being replaced from the stack. The occupied buffers do not contribute to the hit rate. Thus, what really matters is IRR, not recency. To improve LRU, the criterion to determine which accessed blocks are to be cached should be the L blocks with the smallest IRRs, rather than the L blocks with their recencies no more than L (L is the cache size). Following this criterion, the LIRS algorithm uses the LIRS stack to dynamically predict the L blocks that will have the smallest IRRs. The LIRS stack serves two purposes: 1) providing a threshold for being a LIR block and 2) holding the L blocks with the smallest IRRs (i.e., LIR blocks). In the LIRS algorithm, the threshold is $Rmax$, the recency of the LIR block at the LIRS stack bottom. The threshold is also the LIRS stack size.

4.3.1 The Relationship between LIRS Stack Size and Access Characteristics

To get insights into the relationship of the LIRS stack size and workload access characteristics, we plot the ratio of the LIRS stack size and the LRU stack size for two workloads, $postgres$ and $sprite$, in Fig. 8, where we fix the cache size at 500 blocks. We find that the LIRS stack size is an inherent reflection of the LRU capability to exploit locality. If the references have a strong locality, most of the references are to the blocks with small recencies. Thus, the LRU stack still holds these blocks while they get reaccessed and LRU achieves a high hit rate. At the same time, these blocks are low IRR blocks, i.e., most of the references go to the LIR blocks, which would leave only a small number of HIR blocks in the LIRS stack. So, the LIRS stack size is small and close to the LRU stack size. This is the case for workload $sprite$. With 500 buffer blocks, the LRU stack is able to hold the most frequently referenced blocks. On the other hand, LIRS can find enough low IRR blocks within the recency range covered by the LRU stack. So, there is no need for LIRS to significantly raise its stack size to hold a large number of blocks with high recencies in the cache. This is evidenced in Fig. 8 right, where the ratios of the LIRS and LRU stack sizes are not far from 1 for most of the period of time. However, once LIRS cannot find enough low IRR blocks within the size of the LRU stack, it will raise its size accordingly. We observe that the LIRS stack size of $postgres$ is significantly increased in several phases during the periods when more references go to the blocks with high recencies than to those with low recencies. With a cache size of 500 and a fixed stack size, LRU cannot make the locality distinction among the blocks with high recencies and causes their references to all miss. By increasing the stack size according to the current access characteristics, LIRS can make the distinction among blocks with weak locality and make a decision to replace the blocks with a weak locality. The experiments also hint that the LIRS stack size is a good indicator of the LRU-friendliness of a workload.

The 2Q Replacement algorithm also tries to identify blocks of small IRRs and to hold them in cache. It relies on queue $A1out$ to decide whether a block is qualified to be promoted to stack Am so that it can be cached for a long time or, consequently, to decide whether a block in Am

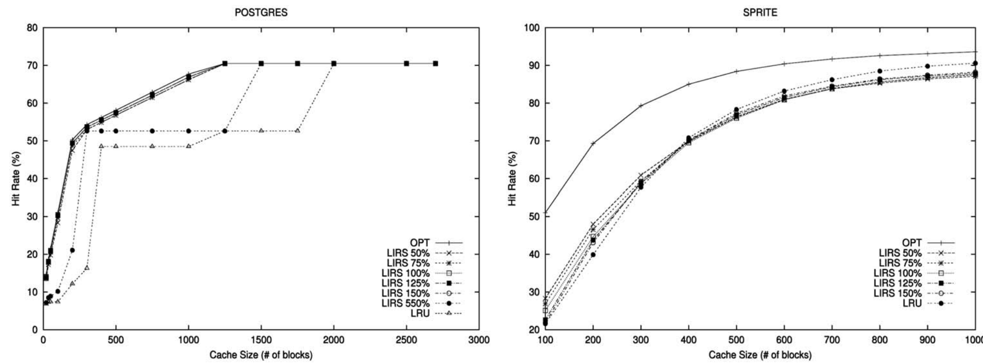


Fig. 9. The hit rate curves of **postgres** and **sprite** by varying the ratio of the status switching threshold and $Rmax$ in LIRS, as well as the curves for OPT and LRU.

should be demoted out of Am. In 2Q, the size of $A1out$ serves as a threshold to identify the blocks of small IRRs and Am holds these blocks. Because the threshold is intended to predict the blocks with the L smallest IRRs among all accessed blocks, 2Q should also consider the access characteristics of blocks in Am. Unfortunately, it does not and only the blocks in $A1out$ are used for setting the threshold. The recommended size of $A1out$ in paper [10] is 50 percent of the cache size. With a fixed threshold, 2Q could make it either too easy or too difficult for the blocks to join in Am with the varying access patterns, This explains why 2Q cannot provide a consistent performance improvement over LRU.

4.3.2 LRU as a Special Member of the LIRS Family

In the LIRS algorithm, the largest recency of the LIR blocks, $Rmax$, serves as a threshold for status switching. An HIR block with a new IRR smaller than the LIRS threshold can change into LIR status and may demote an LIR block into HIR status. The threshold controls how easily an HIR block may become an LIR block or how difficult it is for an LIR block to become an HIR one. We scale the threshold by a weight factor to get insights into the relationship of LRU and LIRS. A weight factor defines a particular LIRS alternative. So, with the scaling, we have a family of LIRS algorithms with various thresholds. Lowering the threshold value, we are able to strengthen the stability of the LIR block set by making it more difficult for HIR blocks to switch their status into LIR. It also prevents the LIRS algorithm from responding to the relatively small IRR variance. Increasing the threshold value, we go in the opposite direction. In this way, LRU becomes a special member of the LIRS family—an LIRS algorithm with an indefinitely large threshold, which always gives any accessed block an LIR status and keeps it in the cache until it is evicted from the stack bottom.

Fig. 9 presents the results of a sensitivity study of the threshold value. We again use workloads $postgres$ and $sprite$ to observe the effects of changing the threshold values from 50 percent, 75 percent, 100 percent, 125 percent to 150 percent of $Rmax$. For $postgres$, we include a very large threshold value—550 percent of $Rmax$ to highlight the relationship between LIRS and LRU. We have two observations. First, LIRS is not sensitive to the threshold value across a large range. In $postgres$, the curves for the threshold values of 100 percent, 125 percent, 150 percent of $Rmax$ are overlapped and the curves for 50 percent, 75 percent of $Rmax$ are slightly lower than the curve for 100 percent of the $Rmax$ threshold. Second, the LIRS algorithm can simulate LRU behavior by significantly increasing the threshold. As the threshold value increases to 550 percent of $Rmax$, the LIRS curve of $postgres$ is very similar to that of LRU in its shape and is close to the LRU curve. Further increasing the threshold value makes the LIRS curve overlaps with the LRU curve. For $sprite$, an LRU-friendly workload, increasing the threshold value makes the LIRS hit rate curve move slowly to the LRU curve.

5 Sensitivity and Overhead Analysis

5.1 Cache Allocation for Resident HIR Blocks

L_{hirs} is the only parameter in the LIRS algorithm. The blocks in the LIR block set can stay in the cache for a longer time than those in the HIR block set and experience fewer page faults. A sufficiently large L_{lirs} (the cache size for LIR blocks) ensures there are a large number of LIR blocks. For this purpose, we set L_{lirs} to be 99 percent of the cache size, L_{hirs} to be 1 percent of the cache size in our experiments, and achieve expected performance. From the other perspective, an increased L_{hirs} may also be beneficial to the performance in some cases: It reduces the first time reference misses. For a large size of stack Q (large L_{hirs}), it is more likely that an HIR will be reaccessed before it is evicted from the stack, which can help the HIR block change into LIR status without experiencing an extra miss. However, the benefit of large L_{hirs} is limited because the number of such kind of misses is small.

We use two workloads, $postgres$ and $sprite$, to observe the effect of changing the size. We change L_{hirs} from two blocks, to 1 percent, 10 percent, 20 percent, and 30 percent of the cache size. Fig. 10 shows the results of the sensitivity study on L_{hirs} for $postgres$ and $sprite$. For each workload, we measure the hit rates of OPT, LRU, and LIRS with different L_{hirs} sizes with increasing cache sizes. We have two observations. First, for both workloads, we find that LIRS is not sensitive to the increase of L_{hirs}. Even for a very large L_{hirs}, which is not in favor of LIRS, the performance of LIRS with different cache sizes is still acceptable. With the increase of L_{hirs}, the hit rates

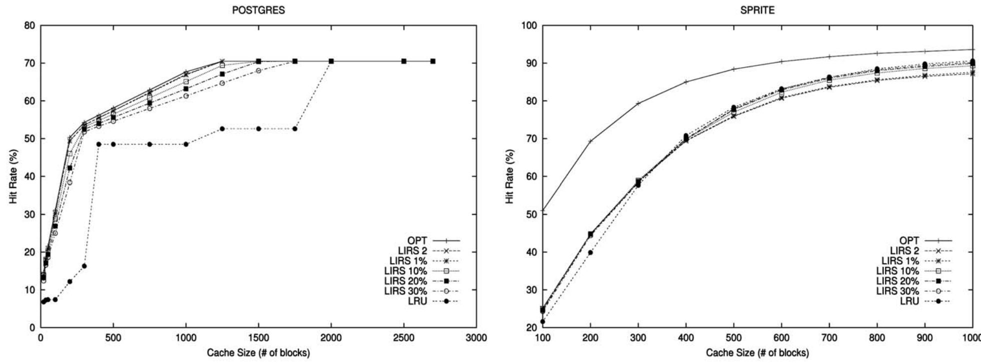


Fig. 10. The hit rate curves of **postgres** and **sprite** by varying the size of stack Q (L_{hirs}) of the LIRS algorithm, as well as the curves for OPT and LRU. "LIRS 2" means the size of Q is 2, "LIRS x%" means the size of Q is x percent of the cache size in blocks.

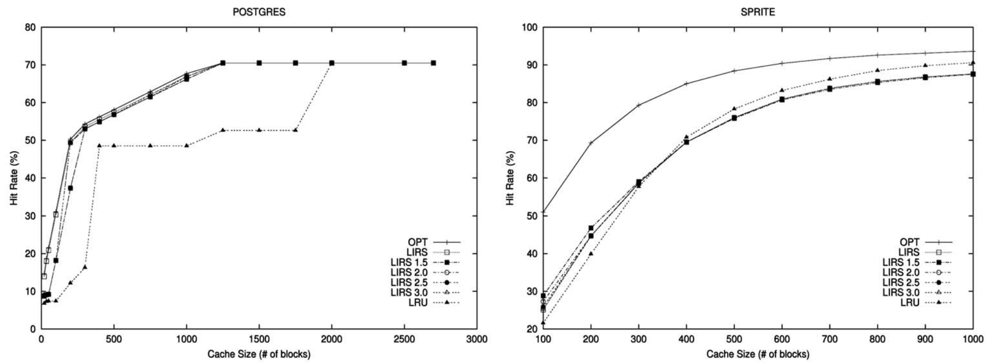


Fig. 11. The hit rate curves of **postgres** and **sprite** by varying the LIRS stack size limit, as well as the curves for OPT and LRU. Limits are represented by ratios of LIRS stack size limit and cache size in blocks.

of LIRS approach those of LRU. Second, our experiments indicate that increasing L_{hirs} reduces the performance benefits of LIRS to workload $postgres$, but slightly improves performance of workload $sprite$.

5.2 Overhead Analysis

LRU is known for its simplicity and efficiency. Comparing the time and space overhead of LIRS and LRU, we show that LIRS keeps the LRU merit of low overhead. The time overhead of LIRS algorithm is $O(1)$, which is almost the same as LRU with a few additional operations such as those on stack Q for resident HIR blocks. The extended portion of the LIRS stack S is the additional space overhead of the LIRS algorithm.

The stack S contains metadata for the blocks with their recency less than $Rmax$. When there is a burst of first-time block references, the LIRS stack could grow to be unacceptably large. Imposing a size limit is a practical issue in the implementation of the LIRS algorithm. In an updated version of LIRS, the LIRS stack has a size limit that is larger than L, and we remove the HIR blocks close to the bottom from the stack once the LIRS stack size exceeds the limit. We have tested a range of small stack size limits, from 1.5 times to 3.0 times of L. From Fig. 11, we can observe that, even with these strict space restrictions, LIRS retains its desirable performance. The effect of limiting LIRS stack size is equivalent to reducing the threshold values in Section 4.3.2. As expected, the results are consistent with the ones presented there. In addition, since a stack entry consists of only several bytes, it is easily affordable to have an LIRS stack size limit much more than three times LRU stack size. There would be little negative effect on LIRS performance by enforcing the limit of such a large size.

6 Conclusions

Replacement algorithms play important roles in the buffer cache management and their effectiveness and efficiency are crucial to the performance of file systems, databases, and other data management systems. We make two contributions in this paper by proposing the LIRS algorithm: 1) We show that LRU limitations with weak locality workloads can be successfully addressed without relying on the explicit access pattern detections. 2) We show earlier work on improving LRU such as LRU-K or 2Q can evolve into one algorithm with consistently superior performance, without tuning or adaptation of sensitive parameters. The effort of these algorithms, which only trace their own history information of each referenced block, is promising to produce an algorithm that is simple and low overhead yet effective for weak locality access patterns. We have shown the LIRS algorithm accomplishes this goal.

As a general-purpose replacement algorithm, the LIRS algorithm also has its potential to be applied in the virtual memory management for its simplicity and its LRU-like assumption on workload characteristics. Because virtual memory system cannot afford an overhead proportional to the number of memory accesses, neither LRU nor LIRS can be directly used there. We have designed an LIRS approximation, called CLOCK-Pro, with a reduced overhead comparable to that of the CLOCK replacement policy

[12]. The results of an implementation of the LIRS approximation in a Linux kernel have shown its significant performance advantages in terms of hit rates and program run times.

ACKNOWLEDGMENTS

This work is supported in part by the US National Science Foundation under grants CCR-9812187 and CCR-0098055. The authors are grateful to Dr. Sam H. Noh at Hong-IK University, Drs. Jong Min Kim, Donghee Lee, and Jongmoo Choi at the Seoul National University for providing us with their traces and simulators. They are also grateful to Dr. Scott Kaplan at Amherst College and Dr. Yannis Smaragdakis at the Georgia Institute of Technology, who provided them with the latest version of their EELRU simulator and traces. The preliminary results of this work were presented in [11].

REFERENCES

[1] L.A. Belady, R.A. Nelson, and G.S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," *Comm. ACM,* vol. 12, pp. 349-353, 1969.

[2] E.G. Coffman and P.J. Denning, *Operating Systems Theory.* Prentice-Hall, 1973.

[3] P. Cao, E.W. Felten, and K. Li, "Application-Controlled File Caching Policies," *Proc. USENIX Summer 1994 Technical Conf.,* pp. 171-182, June 1994.

[4] J. Choi, S. Noh, S. Min, and Y. Cho, "Towards Application/File-Level Characterization of Block References: A Case for Fine-Grained Buffer Management," *Proc. 2000 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 286-295, June 2000.

[5] J. Choi, S. Noh, S. Min, and Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme," *Proc. 1999 Ann. USENIX Technical Conf.,* pp. 239-252, June 1999.

[6] C. Ding and Y. Zhong, "Predicting Whole-Program Locality through Reuse-Distance Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 245-257, June 2003.

[7] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Trans. Database Systems,* pp. 560-595, Dec. 1984.

[8] C. Gniady, A.R. Butt, and Y.C. Hu, "Program Counter Based Pattern Classification in Buffer Caching," *Proc. Sixth Symp. Operating Systems Design and Implementation,* pp. 395-408, Dec. 2004.

[9] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," *Proc. 1997 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 115-126, May 1997.

[10] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. 20th Int'l Conf. Very Large Data Bases,* pp. 439-450, Sept. 1994.

[11] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *Proc. 2002 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 31-42, June 2002.

[12] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," *Proc. 2005 Ann. USENIX Technical Conf.,* pp. 323-336, Apr. 2005.

[13] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, "A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References," *Proc. Fourth Symp. Operating System Design and Implementation,* pp. 119-134, Oct. 2000.

[14] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies," *Proc. 1999 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 134-143, May 1999.

[15] T.C. Mowry, A.K. Demke, and O. Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Application," *Proc. Second USENIX Symp. Operating Systems Design and Implementation,* pp. 3-17, Oct. 1996.

[16] N. Megiddo and D. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *Proc. Second USENIX Conf. File and Storage Technologies,* pp. 115-130, Mar. 2003.

[17] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," *Proc. 1993 ACM SIGMOD Int'l Conf. Management of Data,* pp. 297-306, May 1993.

[18] V. Phalke and B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," *Proc. 1995 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 291-300, May 1995.

[19] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proc. 15th Symp. Operating System Principles,* pp. 1-16, Dec. 1995.

[20] J.T. Robinson and N.V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. 1990 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 134-142, May 1990.

[21] C. Ruemmler and J. Wilkes, "UNIX Disk Access Patterns," *Proc. Usenix Winter 1993 Technical Conf.,* pp. 405-420, Jan. 1993.

[22] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," *Proc. 1999 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems,* pp. 122-133, May 1999.

[23] Y. Zhou, J.F. Philbin, and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," *Proc. 2001 Ann. USENIX Technical Conf.,* pp. 91-104, June 2001.

Song Jiang received the BS and MS degrees in computer science from the University of Science and Technology of China in 1993 and 1996, respectively, and received the PhD degree in computer science from the College of William and Mary in 2004. He is a postdoctoral research associate at the Los Alamos National Laboratory, developing next generation operating systems for high-end systems. He received the S. Park Graduate Research Award from the College of William and Mary in 2003. His research interests are in the areas of operating systems, computer architecture, and distributed systems.

Xiaodong Zhang received the BS degree in electrical engineering from Beijing Polytechnic University in 1982 and the MS and PhD degrees in computer science from the University of Colorado at Boulder in 1985 and 1989, respectively. He is the Lettie Pate Evans Professor of computer science and the department chair at the College of William and Mary. He was the program director of Advanced Computational Research at the US National Science Foundation from 2001 to 2003. He is a past editorial board member of the *IEEE Transactions on Parallel and Distributed Systems* and currently serves as an editorial board member for the *IEEE Transactions on Computers* and an associate editor of *IEEE Micro*. His research interests are in the areas of parallel and distributed computing and systems and computer architecture. He is a senior member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

MORGAN & CLAYPOOL PUBLISHERS

Multi-Core Cache Hierarchies

Rajeev Balasubramonian
Norman Jouppi
Naveen Muralimanohar

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Mark D. Hill, Series Editor

BALASUBRAMONIAN, JOUPPI, MURALIMANOHAR
MULTI-CORE CACHE HIERARCHIES
MORGAN & CLAYPOOL

Multi-Core Cache Hierarchies

Synthesis Lectures on Computer Architecture

Editor
Mark D. Hill, ***University of Wisconsin***

Synthesis Lectures on Computer Architecture publishes 50- to 100-page publications on topics pertaining to the science and art of designing, analyzing, selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. The scope will largely follow the purview of premier computer architecture conferences, such as ISCA, HPCA, MICRO, and ASPLOS.

Multi-Core Cache Hierarchies
Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar
2011

A Primer on Memory Consistency and Cache Coherence
Daniel J. Sorin, Mark D. Hill, and David A. Wood
2011

Dynamic Binary Modification: Tools, Techniques, and Applications
Kim Hazelwood
2011

Quantum Computing for Computer Architects, Second Edition
Tzvetan S. Metodi, Arvin I. Faruque, and Frederic T. Chong
2011

High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities
Dennis Abts and John Kim
2011

Processor Microarchitecture: An Implementation Perspective
Antonio González, Fernando Latorre, and Grigorios Magklis
2010

Transactional Memory, 2nd edition
Tim Harris, James Larus, and Ravi Rajwar
2010

Computer Architecture Performance Evaluation Methods
Lieven Eeckhout
2010

Introduction to Reconfigurable Supercomputing
Marco Lanzagorta, Stephen Bique, and Robert Rosenberg
2009

On-Chip Networks
Natalie Enright Jerger and Li-Shiuan Peh
2009

The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It
Bruce Jacob
2009

Fault Tolerant Computer Architecture
Daniel J. Sorin
2009

The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines free access
Luiz André Barroso and Urs Hölzle
2009

Computer Architecture Techniques for Power-Efficiency
Stefanos Kaxiras and Margaret Martonosi
2008

Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency
Kunle Olukotun, Lance Hammond, and James Laudon
2007

Transactional Memory
James R. Larus and Ravi Rajwar
2006

Quantum Computing for Computer Architects
Tzvetan S. Metodi and Frederic T. Chong
2006

Copyright © 2011 by Morgan & Claypool

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews, without the prior permission of the publisher.

Multi-Core Cache Hierarchies

Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar

www.morganclaypool.com

ISBN: 9781598297539 paperback
ISBN: 9781598297546 ebook

DOI 10.2200/S00365ED1V01Y201105CAC017

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Lecture #17
Series Editor: Mark D. Hill, *University of Wisconsin*
Series ISSN
Synthesis Lectures on Computer Architecture
Print 1935-3235 Electronic 1935-3243

Multi-Core Cache Hierarchies

Rajeev Balasubramonian
University of Utah

Norman P. Jouppi
HP Labs

Naveen Muralimanohar
HP Labs

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #17

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

A key determinant of overall system performance and power dissipation is the cache hierarchy since access to off-chip memory consumes many more cycles and energy than on-chip accesses. In addition, multi-core processors are expected to place ever higher bandwidth demands on the memory system. All these issues make it important to avoid off-chip memory access by improving the efficiency of the on-chip cache. Future multi-core processors will have many large cache banks connected by a network and shared by many cores. Hence, many important problems must be solved: cache resources must be allocated across many cores, data must be placed in cache banks that are near the accessing core, and the most important data must be identified for retention. Finally, difficulties in scaling existing technologies require adapting to and exploiting new technology constraints.

The book attempts a synthesis of recent cache research that has focused on innovations for multi-core processors. It is an excellent starting point for early-stage graduate students, researchers, practitioners who wish to understand the landscape of recent cache research. The book is suitable as a reference for advanced computer architecture classes as well as for experienced researchers and VLSI engineers.

KEYWORDS

computer architecture, multi-core processors, cache hierarchies, shared and private caches, non-uniform cache access (NUCA), quality-of-service, cache partitions, replacement policies, memory prefetch, on-chip networks, memory cells.

To our highly supportive families and colleagues.

Contents

Preface xi

Acknowledgments xv

1 Basic Elements of Large Cache Design 1

1.1 Shared Vs. Private Caches 1
1.1.1 Shared LLC 2
1.1.2 Private LLC 4
1.1.3 Workload Analysis 6
1.2 Centralized Vs. Distributed Shared Caches 7
1.3 Non-Uniform Cache Access 10
1.4 Inclusion 13

2 Organizing Data in CMP Last Level Caches 15

2.1 Data Management for a Large Shared NUCA Cache 15
2.1.1 Placement/Migration/Search Policies for D-NUCA 16
2.1.2 Replication Policies in Shared Caches 23
2.1.3 OS-based Page Placement 25
2.2 Data Management for a Collection of Private Caches 34
2.3 Discussion 40

3 Policies Impacting Cache Hit Rates 41

3.1 Cache Partitioning for Throughput and Quality-of-Service 41
3.1.1 Introduction 41
3.1.2 Throughput 43
3.1.3 QoS Policies 52
3.2 Selecting a Highly Useful Population for a Large Shared Cache 56
3.2.1 Replacement/Insertion Policies 56
3.2.2 Novel Organizations for Associativity 64
3.2.3 Block-Level Optimizations 66
3.3 Summary 76

4 Interconnection Networks within Large Caches **79**

4.1 Basic Large Cache Design 79

4.1.1 Cache Array Design 79

4.1.2 Cache Interconnects 80

4.1.3 Packet-Switched Routed Networks 81

4.2 The Impact of Interconnect Design on NUCA and UCA Caches 89

4.2.1 NUCA Caches 89

4.2.2 UCA Caches 92

4.3 Innovative Network Architectures for Large Caches 94

5 Technology **101**

5.1 Static-RAM Limitations 101

5.2 Parameter Variation 102

5.2.1 Modeling Methodology 103

5.2.2 Mitigating the Effects of Process Variation 103

5.3 Tolerating Hard and Soft Errors 106

5.4 Leveraging 3D Stacking to Resolve SRAM Problems 108

5.5 Emerging Technologies 110

5.5.1 3T1D RAM 111

5.5.2 Embedded DRAM 113

5.5.3 Non-Volatile Memories 113

6 Concluding Remarks **117**

Bibliography **119**

Authors' Biographies **137**

Preface

The multi-core revolution is well under-way. The first few mainstream multi-core processors appeared around 2005. Today, it is nearly impossible to buy a desktop or laptop that has just a single core in it. The trend is obvious; the number of cores on a chip will likely double every two or three years. Such processor chips will be widely used in the high-performance computing domain: in supercomputers, servers, and high-end desktops. Just as the volume of low-end devices (for example, smartphones) is expected to increase, the volume of high-end devices (servers in datacenters) is also expected to increase. The latter trend is likely because users will increasingly rely on the *"cloud"* for data storage and computation.

For many decades, one of the key determinants of overall system performance has been the memory hierarchy. Access to off-chip memory consumes many cycles and many units of energy. The more data that can be accommodated and found in the caches of a processor chip, the higher the performance and energy efficiency. This continues to be true in the multi-core era. In fact, multi-core processors are expected to place even higher pressure on the memory system: the number of pins on a chip is expected to remain largely constant while the number of cores that must be fed with data is expected to rise sharply. This makes it even more important to minimize off-chip accesses.

Memory hierarchy efficiency is a strong function of the access latencies of on-chip caches and their hit rates. Future last-level caches (LLCs) are expected to occupy half the processor chip's die area and accommodate many mega-bytes of data. The LLC will likely be composed of many banks scattered across the chip. Access to data will require navigation of long wires and traversal through multiple routing elements. Each access will therefore require many tens of cycles of latency and many nanojoules of energy, depending on the distance that must be traveled. Cache resources will have to be allocated across threads and parts of the LLC may be private to a thread while other parts may be shared by multiple threads.

As a result, caching techniques will undergo evolution in the coming years because of new challenges imposed by multi-core platforms and workloads. Cache policies must now worry about interference among threads as well as large and non-uniform latencies and energy for data transmission between cache banks and cores. On-chip non-local wires continue to scale poorly, increasing the role of the interconnect during cache access. It is therefore imperative that we (i) devise caching policies that reduce long-range communication and (ii) create low-overhead networks to better handle long-range communication when it is required. Several new technology phenomena will also require innovation within the caches. These include the emergence of parameter variation, hard and soft error rates, leakage energy in caches, and thermal constraints from 3D stacking.

Consider the following examples of the game-changing impact of multi-core on caching policies. After years of reliance on LRU-like policies for cache replacement, several papers have

emerged in recent years that have shown that alternative approaches are much more effective for replacement in multi-core LLCs. Likewise, the past decade has seen many papers that consider variations of private and shared LLCs, attempting to combine the best of both worlds. Hence, the past and upcoming decades are exciting times for cache research. In retrospect, it should have been obvious that multi-core processors would be imminent; many papers in the 1990s had pointed to this trend. Yet, overall, the community was a little slow to embrace multi-core research. As a result, the pace of multi-core research saw an acceleration only after the arrival of the first commercial multi-core processors. Much work remains, especially for the memory hierarchies of future many-core processors.

Book Organization

The goal of this book is to synthesize much of the recent cache research that has focused on innovations for multi-core processors. For any researcher or practitioner that wishes to understand the landscape of recent cache work, we hope that the book will be an ideal starting point. We also expect early-stage graduate students to benefit from such a synthesis lecture. The book should also serve as a good reference book for advanced computer architecture classes. We expect that the material here will be accessible to both computer scientists and VLSI engineers. The book is not intended as a substitute to reading relevant full papers and chasing down older references. The book will hopefully improve one's breadth and awareness of a multitude of caching topics, while making research on a specific topic more efficient.

Given the vastness of the memory hierarchy topic, we had to set some parameters for what would be worthy of inclusion in this book. We have primarily focused on recent work (2004 and after) that has a strong connection with the use of multiple cores. We have focused our coverage on papers that appear at one of the four primary venues for architecture research: ISCA, MICRO, ASPLOS, and HPCA. However, the book has several discussions of papers that have appeared at other venues and that have made a clear impact within the community. In spite of our best efforts, we have surely left out a few papers that deserve mention; we can hopefully correct some of our oversights in subsequent versions. We encourage readers to contact us to point out our omissions.

The area of multi-core caching has a strong overlap with several other areas within computer architecture. We have explicitly left some of these areas out of this book because they have been covered by other synthesis lectures:

- Off-chip memory systems [10]
- On-chip network designs [11]
- Core memory components (load-store-queue, L1 cache) [12]
- Cache coherence and consistency models [13]

- Phase change memory [14]
- Power optimizations [15]

The discussions in the book have attempted to highlight the key ideas in papers. We have attempted to convey the novelty and the qualitative contribution of each paper. We have typically not summarized the quantitative improvements of each idea. We realize that the mention of specific numbers from papers may be misleading as each paper employs different benchmarks and simulation infrastructure parameters.

The second chapter provides background and a taxonomy for multi-core cache hierarchies. The third chapter then examines policies that bridge the gap between shared LLCs and private LLCs. The papers in Chapter 2 typically assume that the LLC is made up of a collection of banks, with varying latencies to reach each bank. The considered policies attempt to place data in these banks so that access latency is minimal and hit rates can be maximized. Chapter 3 also focuses on hit rate optimization, but it does so for a single cache bank. Instead of moving data between banks, the considered policies improve hit rates with better replacement policies, better organizations for associativity, and block-level optimizations (prefetch, dead block prediction, compression, etc.). That chapter also examines how a single cache bank can be partitioned among multiple threads for high throughput and quality-of-service. Since access to an LLC often requires navigation of an on-chip network, Chapter 4 describes on-chip network innovations that have a strong interaction with caching policies. Chapter 5 describes how modern technology trends are likely to impact the design of future caches. It covers modern technology phenomena such as 3D die-stacking, parameter variation, rising error rates, and emerging non-volatile memories. Chapter 6 concludes with some thoughts on avenues for future work.

Figure 1 uses the same classification as above and shows the number of papers that have appeared in each cache topic in the past seven years at the top four architecture conferences. Note that there can be multiple ways to classify the topic of a paper, and the data should be viewed as being approximate. The data serves as an indicator of hot topics within multi-core caching. Activity appears to be highest in technology phenomena, block prefetch, and shared caches.

Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar
May 2011

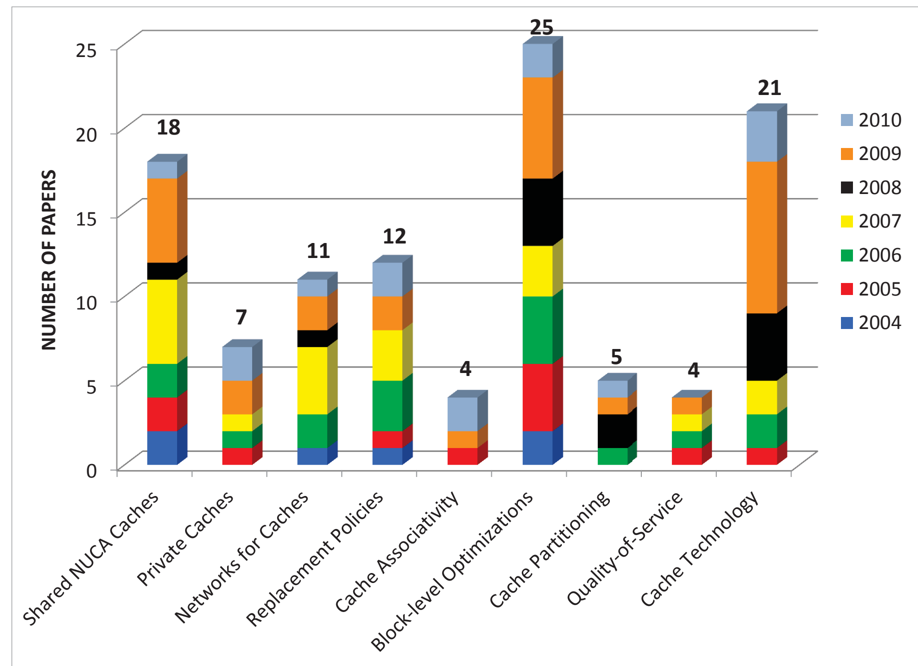


Figure 1: Number of papers in various cache topics in the last seven years at ISCA, MICRO, ASPLOS, and HPCA.

Acknowledgments

We thank everyone that provided comments and feedback on early drafts of this synthesis lecture, notably, Mark Hill, Aamer Jaleel, Gabriel Loh, Mike Morgan, and students in the Utah Arch research group.

Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar
May 2011

CHAPTER 1

Basic Elements of Large Cache Design

This chapter presents a landscape of cache hierarchy implementations commonly employed in research/development and identifies their key distinguishing features. These features include the following: shared vs. private, centralized vs. distributed, and uniform vs. non-uniform access. There is little consensus in the community about what constitutes an optimal cache hierarchy implementation. Some levels of the cache hierarchy employ private and uniform access caches, while other levels employ shared and non-uniform access. We will point out the pros and cons of selecting each feature, and it is perfectly reasonable for a research effort to pick any combination of features for their baseline implementation. Much of the focus of this book is on the design of the on-chip *Last-Level Cache (LLC)*. In the past, most on-chip cache hierarchies have been comprised of two levels (L1 and L2), but it is becoming increasingly common to incorporate three levels in the cache hierarchy (L1, L2, and an L3 LLC). As we explain in this chapter and the next, future LLCs have a better chance of optimizing miss rates, latency, and complexity if they are implemented as shared caches. This chapter also discusses other basics that are required to understand modern cache innovations.

Before getting started, a couple of terminology clarifications are in order. In a cache hierarchy, a cache level close to the processor is considered an "upper-level" cache, while a cache level close to main memory is considered a "lower-level" cache. We will also interchangeably use the terms "cache line" and "cache block", both intended to represent the smallest unit of data handled during cache fetch and replacement.

1.1 SHARED VS. PRIVATE CACHES

Most modern high-performance processors incorporate multiple levels of the cache hierarchy within a single chip. In a multi-core processor, each core typically has its own private L1 data and L1 instruction caches. Considering that every core must access the L1 caches in nearly every cycle, it is not typical to have a single L1 cache (either data or instruction cache) shared by multiple cores. A miss in the L1 cache initiates a request to the L2 cache. For most of the discussion, we will assume that the L2 is the LLC. But the same arguments will also apply to an L3 LLC in a 3-level hierarchy, where the L1 and L2 are private to each core. We first compare the properties of shared and private LLCs.

1.1.1 SHARED LLC

A single large L2 LLC may be shared by multiple cores on a chip. Since the requests originating from a core are filtered by its L1 caches, it is possible for a single-ported L2 cache to support the needs of many cores. One example organization is shown in Figure 1.1. In this design, eight cores

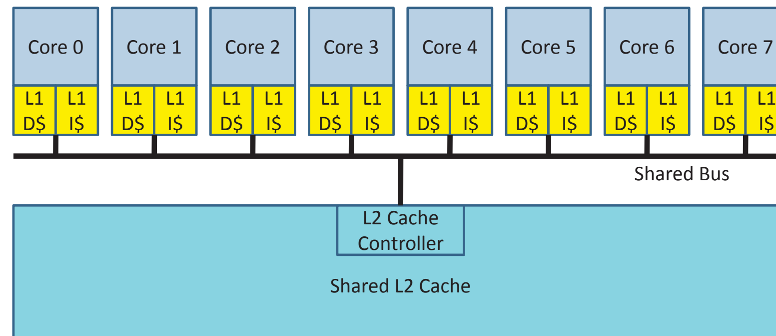


Figure 1.1: Multi-core cache organization with a large shared L2 cache and private L1 caches per core. A snooping-based cache coherence protocol is implemented with a bus connecting the L1s and L2.

share a single large L2 cache. When a core fails to find data in its L1 caches, it places the request on a bus shared by all cores. The L2 cache controller picks requests off this bus and performs the necessary look-up. In such a large shared L2 cache, there are no duplicate copies of a memory block, but a given block may be cached in multiple different L1 caches. Coherence must be maintained among the L1s and the L2. In the bus-based example in Figure 1.1, coherence is maintained with a snooping-based protocol. Assume that the L1 caches employ a write-back policy. When a core places a request on the bus, every other core sees this request and looks up its L1 cache to see if it has a copy of the requested block. If a core has a copy of the block in modified state, *i.e.*, this copy happens to be the most up-to-date version and the only valid copy of the block, the core must respond by placing the requested data on the bus. If no core has the block in modified state, the L2 cache must provide the requested data. The L2 cache controller figures out that it must respond by examining a set of control signals that indicate that the cores have completed their snoops and do not have the block in modified state. If the requesting core is performing a write, copies of that block in other L1 caches are invalidated during the snoop operation. Of course, there can be many variations of this basic snooping-based protocol [16, 17]. If a write-through policy is employed for the L1 caches, an L1 miss is always serviced by the L2. A write-through policy can result in significant bus traffic and energy dissipation; this overhead is not worthwhile in the common case. Similarly, write-update cache coherence protocols are also more traffic intensive and not in common use. However, some of

these design guidelines are worth re-visiting in the context of modern single-chip multi-cores with relatively cheap interconnects.

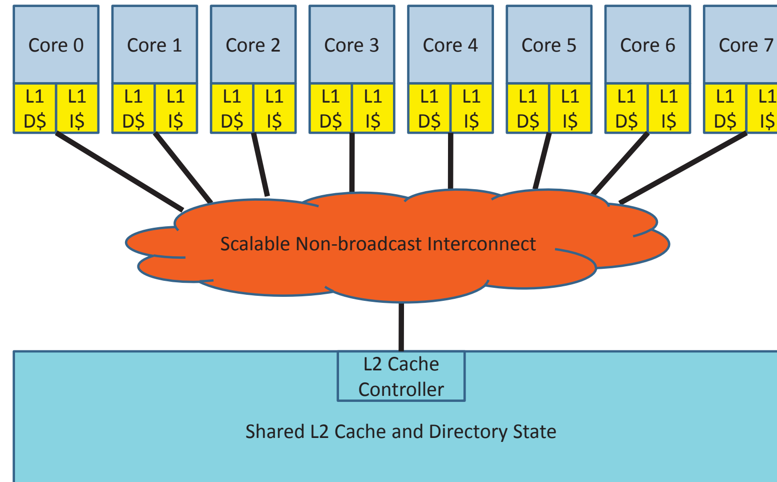


Figure 1.2: Multi-core cache organization with a large shared L2 cache and private L1 caches per core. A scalable network connects the L1 caches and L2 and a directory-based cache coherence protocol is employed. Each block in the L2 cache maintains directory state to keep track of copies cached in L1.

The above example primarily illustrates the interface required before accessing a shared cache. In essence, a mechanism is required to ensure coherence between the shared L2 and multiple private L1s. If the number of cores sharing an L2 is relatively small (16 or fewer), a shared bus and a snooping-based coherence protocol will likely work well. For larger-scale systems, a scalable interconnect and a directory-based coherence protocol are typically employed. As shown in Figure 1.2, the cores and the L2 cache are connected with some scalable network and broadcasting a request is no longer an option. The core sends its request to the L2 cache and each L2 block is associated with a directory that keeps track of whether other L1 caches have valid copies of that block. If necessary, other caches are individually contacted to either invalidate data or obtain the latest copy of data.

There are many advantages to employing a shared cache. First, the available storage space can be dynamically allocated among multiple cores, leading to better utilization of the overall cache space. Second, if data is shared by multiple cores, only a single copy is maintained in L2, again leading to better space utilization and better cache hit rates. Third, if data is shared by multiple cores and subject to many coherence misses, the cache hierarchy must be navigated until the coherence interface and shared cache is encountered. The sooner a shared cache is encountered, the sooner coherence misses can be resolved.

The primary disadvantages of a shared cache are as follows. The working sets of different cores may interfere with each other and impact each other's miss rates, possibly leading to poorer quality-of-service. As explained above, access to a shared L2 requires navigation of the coherence interface: this may impose overheads if the cores are mostly dealing with data that is not shared by multiple cores. Finally, many papers cite that a single large shared L2 cache may have a relatively long access time on average. Also, a core may experience many contention cycles when attempting to access a resource shared by multiple cores. However, we will subsequently (Section 2.1) show that both of these disadvantages can be easily alleviated.

It must be noted that many of our examples assume that the L2 cache is inclusive, *i.e.*, if a data block is present in L1, it is necessarily also present in L2. In Section 1.4, we will discuss considerations in selecting inclusive and non-inclusive implementations.

1.1.2 PRIVATE LLC

A popular alternative to the single shared LLC is a collection of private last-level caches. Assuming a two-level hierarchy, a core is now associated with private L1 instruction and data caches *and* a private unified (handling data and instructions) L2 cache. A miss in L1 triggers a look-up of the core's private L2 cache. Some of the advantages/disadvantages of such an organization are already apparent.

The working sets of threads executing on different cores will not cause interference in each other's L2 cache. Each private L2 cache is relatively small (relative to a single L2 cache that must be shared by multiple cores), allowing smaller access times on average for L2 hits. The private L2 cache can be accessed without navigating the coherence interface and without competition for a shared resource, leading to performance benefits for threads that primarily deal with non-shared data.

A primary disadvantage of private L2 caches is that a data block shared by multiple threads will be replicated in each thread's private L2 cache. This replication of data blocks leads to a lower effective combined L2 cache capacity, relative to a shared L2 cache of similar total area. In other words, four private 256 KB L2 caches will accommodate less than 1 MB worth of data because of duplicate copies of a block, while a 1 MB shared L2 cache can indeed accommodate 1 MB worth of data. Another disadvantage of a private L2 cache organization is the static allocation of L2 cache space among cores. In the above example, each core is allocated a 256 KB private L2 cache even though some cores may require more or less. In a 1 MB shared L2 cache, it is possible for one core to usurp (say) 512 KB of the total space if it has a much larger working set size than threads on the other cores.

By employing private L2 caches, the coherence interface is pushed down to a lower level of the cache hierarchy. First, consider a small-scale multi-core machine that employs a bus-based snooping coherence protocol. On an L2 miss, the request is broadcast on the bus. Other private L2 caches perform snoop operations and place their responses on the bus. If it is determined that none of the other private L2 caches can respond, a controller forwards this request to the next level of the hierarchy (either an L3 cache or main memory). When accessing shared data, such a private L2

organization imposes greater latency overheads than a model with private L1s and a shared L2. The key differentiating overheads are the following: (i) the private L2 cache is looked up before placing the request on the bus, (ii) snoops take longer as a larger set of tags must be searched, and (iii) it takes longer to read data out of another large private L2 data array (than another small private L1 cache).

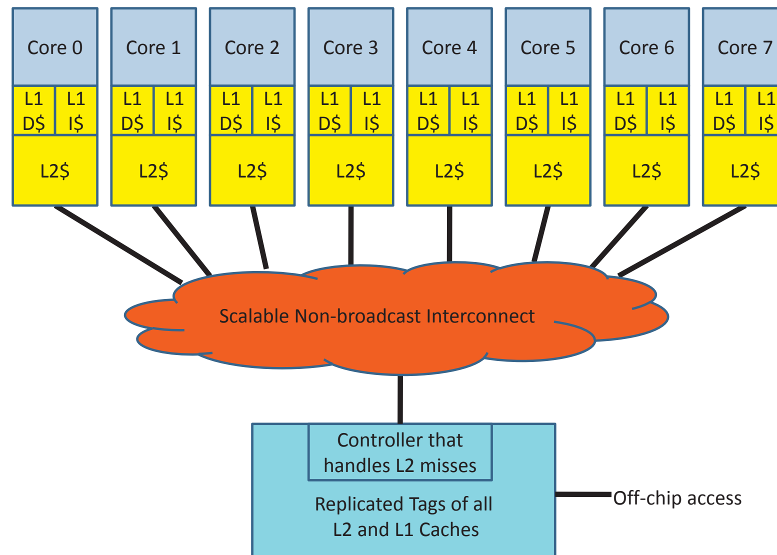


Figure 1.3: Multi-core cache organization where each core has a private L2 cache and coherence is maintained among the private L2 caches with a directory-based protocol across a scalable non-broadcast interconnect.

The coherence interface is even more complex if a directory-based protocol is employed (shown in Figure 1.3). On an L2 miss, the request cannot be broadcast to all cores, but it is sent to a directory. This directory may be centralized or distributed, but in either case, long on-chip distances may have to be traversed. This directory must keep track of all blocks that are cached on chip and it essentially replicates the tags of all the private L2 caches. A highly-associative search is required to detect if the requested address is in any of the private L2 caches. If each of the four 256 KB private L2 caches is 4-way set-associative, the directory look-up will require 16 tag comparisons to determine the state of the block. L1 tags need not be replicated by preserving inclusion between the L1s and L2. If a block is detected in another private L2 cache, messages are exchanged between the directory and cores to move the latest copy of data to the requesting core's private cache. On the other hand, the shared L2 cache simply associates the directory with the unique copy of the block in

L2, thus eliminating the need to replicate L2 tags. In short, assuming inclusion, the use of a shared on-chip LLC makes it easier to detect if a cached copy exists on the chip.

Table 1.1: A comparison of the advantages and disadvantages of private and shared cache organizations.

Shared L2 Cache	Private L2 Caches
No replication of shared blocks (higher effective capacity)	Replication of shared blocks (lower effective capacity)
Dynamic allocation of space among threads/cores (higher effective capacity)	Design-time allocation of space among cores (lower effective capacity)
Quick traversal through coherence interface (low latency for shared data)	Slower traversal through coherence interface (high latency for shared data)
No L2 tag replication for directory implementation (low area requirements)	Directory implementation requires replicated L2 tags (high area requirements)
Higher interference between threads (negatively impacts QoS)	No interference between threads (positively impacts QoS)
Longer wire traversals on average to detect an L2 hit (high average hit latency[1])	Short wire traversals on average to detect an L2 hit (low average hit latency)
High contention when accessing shared resource (bus and L2) (high hit latency for private data)	No contention when accessing L2 cache (low hit latency for private data)

The differences between private and shared L2 cache organizations are summarized in Table 1.1. It is also worth noting that future processors may employ combinations of private and shared caches. For example (Figure 1.4), in a 16-core processor, each cluster of four cores may share an L2 cache, and there are four such L2 caches that are each private to their cluster of four cores. Snooping-based coherence is first maintained among the four L1 data caches and L2 cache in one cluster; snooping-based coherence is again maintained among the four private L2 caches.

1.1.3 WORKLOAD ANALYSIS

Some recent papers have focused on analyzing the impact of baseline shared and private LLCs on various multi-threaded workloads. The work of Jaleel et al. [18] characterizes the behavior of bioinformatics workloads. They show that more than half the cache blocks are shared, and a vast majority of LLC accesses are to these shared blocks. Given this behavior, a shared LLC is a clear

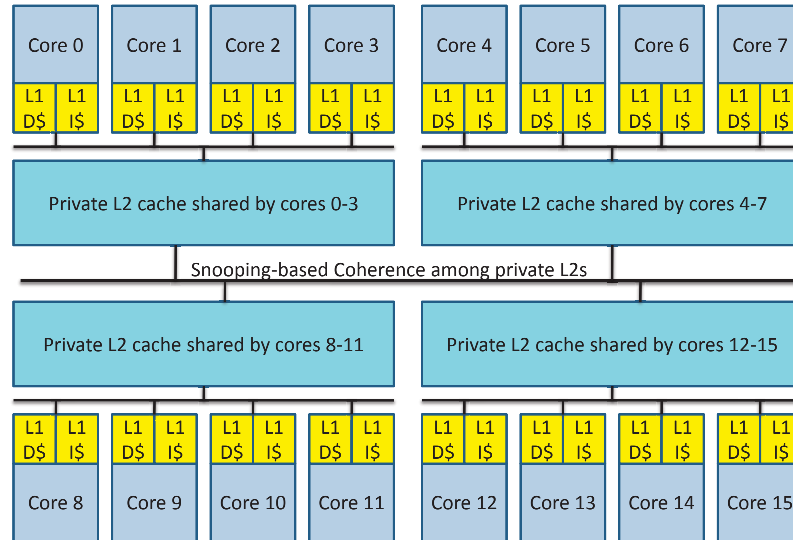


Figure 1.4: 16-core machine where each cluster of four cores has a private L2 cache that is shared by its four cores. There are two hierarchical coherence interfaces here: one among the L1 data caches and L2 cache within a cluster and one among the four private L2 caches.

winner over an LLC that is composed of many private LLCs. Bienia et al. [19] show a workload analysis of the SPLASH-2 and PARSEC benchmark suites, including cache miss rates and the extent of data sharing among threads. Many other cache papers also report workload characterizations in their analysis, most notably, the work of Beckmann et al. [20] and Hardavellas et al. [3].

1.2 CENTRALIZED VS. DISTRIBUTED SHARED CACHES

This section primarily discusses different implementations for a shared last level cache. At the end of the section, we explain how some of these principles also apply to a collection of private caches.

We have already considered two shared L2 cache organizations in Figures 1.1 and 1.2. In both of these examples, the L2 cache and its controller are represented as a single centralized entity. When an L1 miss is generated, the centralized L2 cache controller receives this request either from the bus (Figure 1.1) or from a link on the scalable network (Figure 1.2). It then proceeds to locate the corresponding block within the L2 cache structure. If the L2 cache is large (as is usually the case), it is itself partitioned into numerous banks, and some sort of interconnection network must be navigated to access data within one of the banks (more details on this in later sections). Thus, some form of network fabric may have to be navigated to simply reach the centralized L2 cache controller and yet another fabric is navigated to reach the appropriate bank within the L2 cache. Depending

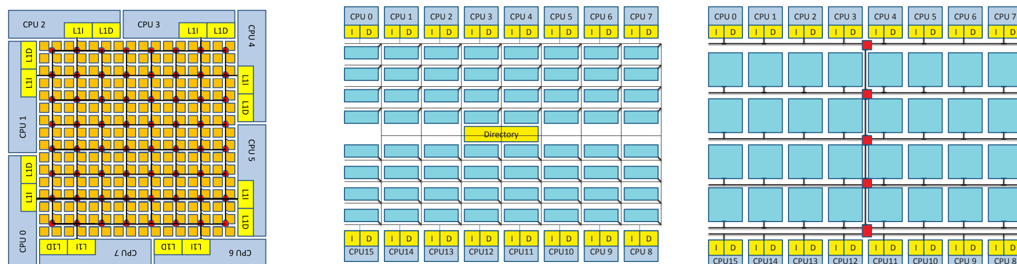


Figure 1.5: Shared L2 cache with a centralized layout (the L2 cache occupies a contiguous area in the middle of the chip and is surrounded by cores). An on-chip network is required to connect the many cache banks to each other and the cores. In all these cases, the functionality of the cache controller is replicated in each of the banks to avoid having to go through a central entity.

on the types of fabrics employed, it may be possible to merge the two. In such a scenario, a single fabric is navigated to directly send the request from a core to the L2 bank that stores the block. The L2 bank will now need some logic to take care of the necessary coherence operations; in other words, the functionality of the L2 cache controller is replicated in each of the banks to eliminate having to go through a single centralized L2 cache controller.

Some example physical layouts of such centralized shared L2 caches are shown in Figure 1.5. Each of these layouts has been employed in research evaluations (for example, [2, 7, 21, 22]). Even though the cache is banked and the controller functionality is distributed across the banks, we will refer to these designs as *Centralized* because the LLC occupies a contiguous area on the chip. Such centralized cache structures attempt to provide a central pool of data that may be quickly and efficiently accessed by cores surrounding it. By keeping the cache banks in close proximity to each other, movement of data between banks (if required) is simplified. The interconnects required between L2 cache banks and the next level of the hierarchy (say, the on-chip memory controller) are simplified by aggregating all the cache banks together. The L2 cache also ends up being a centralized structure if it is implemented on a separate die that is part of a 3D-stacked chip (assuming a single die-to-die bus that communicates requests and responses between CPUs and a single L2 cache controller).

An obvious extension to this model is the *distributed shared L2 cache*. Even though the L2 cache is logically a shared resource, it may be physically distributed on chip, such that one bank of the L2 may be placed in close proximity to each core. The core, its L1 caches, and one bank (or *slice*) of the L2 cache together constitute one *tile*. A single on-chip network is used to connect all the tiles. When a core has an L1 miss, its request is routed via the on-chip network to the tile that is expected to have the block in its L2 bank. Such a tiled and distributed cache organization is desireable because it allows manufacturers to design a single tile and instantiate as many tiles as allowed by the area budget. It therefore lends itself better to scalable design/verification cost, easy

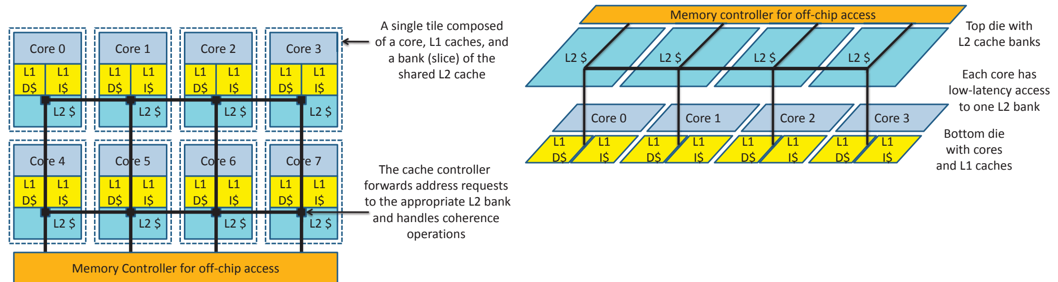


Figure 1.6: A shared L2 cache with a physically distributed layout. One bank (or *"slice"*) of L2 cache is associated with each core. A *"tile"* is composed of a core, its L1 caches, and its associated L2 bank. The figure on the right physically separates the L2 banks onto a separate die in a 3D stack, but it retains the same logical organization as the figure on the left.

manufacture of families of processors with varying numbers of tiles, and simple upgrades to new technology generations. Such distributed shared caches have also been implemented in recent Tilera multi-core processors. Two examples of such a physical layout are shown in Figure 1.6. We will also subsequently see how architectural mechanisms for data placement can take advantage of such a physical organization. The primary disadvantage of this organization is the higher cost in moving data/requests between L2 cache banks and the next level of the memory hierarchy.

While a centralized L2 cache structure may be a reasonable design choice for a processor with a medium number of cores, it may prove inefficient for a many-core processor. Thermal and interconnect (scalability and wire-length) limitations may prevent many cores from surrounding a single large centralized L2 cache. It is therefore highly likely that many-core processors will employ a distributed L2 cache where every core at least has very quick access to one bank of the shared L2 cache. Distributing the L2 cache also has favorable implications for power density and thermals.

Our definitions of *Centralized* and *Distributed* caches are only meant to serve as an informal guideline when reasoning about the properties of caches. A centralized cache is defined as a cache that occupies a contiguous area on the chip. A distributed cache is defined as a cache where each bank is tightly coupled to a core or collection of cores. In Figure 1.6(a), if the layout of cores 4-7 was a mirror image of the cores 0-3, the cache would be classified as both centralized and distributed, which is admittedly odd.

When implementing a private L2 cache organization, it makes little sense to place the core's private L2 cache anywhere but in close proximity to the core. Hence, a private L2 cache organization has a physical layout that closely resembles that of the distributed shared L2 cache just described. In other words, for both organizations, a tile includes a bank of L2 cache; it is the logical policies for placing and managing data within the many L2 banks that determines if the L2 cache space is shared or private.

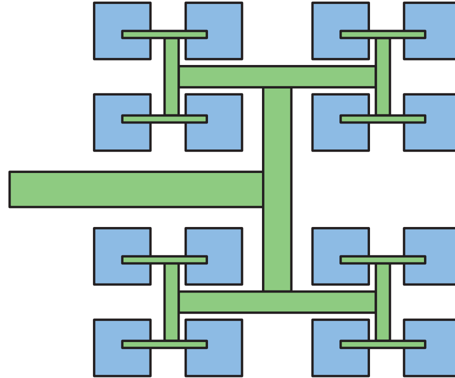


Figure 1.7: An example H-tree network for a uniform cache access (UCA) architecture with 16 arrays.

For the private L2 cache organization, if a core has a miss in its private L2 cache, the request must be forwarded to the coherence interface. If we assume directory-based coherence, the request is forwarded to the directory structure that could itself be centralized or distributed. The considerations in implementing a centralized/distributed on-chip directory are very similar to those in implementing a centralized/distributed shared L2 cache. The on-chip network is therefore employed primarily to deal with coherence operations and accesses to the next level of the hierarchy. The on-chip network for the distributed shared L2 cache is also heavily employed for servicing L2 hits.

1.3 NON-UNIFORM CACHE ACCESS

In most processors until very recently, a cache structure is designed to have a uniform access time regardless of the block being accessed. In other words, the delay for the cache access is set to be the worst-case delay for any block. Such *Uniform Cache Access (UCA)* architectures certainly simplify any associated instruction scheduling logic, especially if the core pipeline must be aware of cache hit latency. However, as caches become larger and get fragmented into numerous banks, there is a clear inefficiency in requiring that every cache access incur the delay penalty of accessing the furthest bank. Simple innovations to the network fabric can allow a cache to support non-uniform access times; in fact, many of the banked cache organizations that we have discussed so far in this chapter are examples of *Non-Uniform Cache Access (NUCA)* architectures.

A UCA banked cache design often adopts an H-tree topology for the interconnect fabric connecting the banks to the cache controller. With an H-tree topology (example shown in Figure 1.7), every bank is equidistant from the cache controller, thus enabling uniform access times to every bank. In their seminal paper [1], Kim et al. describe the innovations required to support

a NUCA architecture and quantify its performance benefits. Firstly, variable access times for the L2 are acceptable as the core pipeline does not schedule instructions based on expected L2 access time. Secondly, instead of adopting an H-tree topology, a grid topology is employed to connect banks to the cache controller. The latency for a bank is a function of its size and the number of network hops required to route the request/data between the bank and the cache controller. It is worth noting that messages on this grid network can have a somewhat irregular pattern, requiring complex mechanisms at every hop to support routing and flow control. These mechanisms were not required in the H-tree network, where requests simply radiated away from the cache controller in a pipelined fashion. The complexity in the network is the single biggest price being paid by a NUCA architecture to provide low-latency access to a fraction of cached data. It can be argued that most future architectures will anyway require complex on-chip networks to handle somewhat arbitrary messaging between the numerous cores and cache banks. Especially in tiled architectures such as the ones shown in Figure 1.6, it is inevitable that the different cache banks incur variable access latencies as a function of network distance.

In the next chapter, we will discuss innovations to NUCA architectures that attempt to cleverly place data in an "optimal" cache bank. These innovations further drive home the point that traffic patterns are somewhat arbitrary and require complex routing/flow control mechanisms. In Chapter 4, we describe on-chip network innovations that are applicable to specific forms of NUCA designs. In the rest of this section, we describe the basic NUCA designs put forth by Kim et al. in their paper [1].

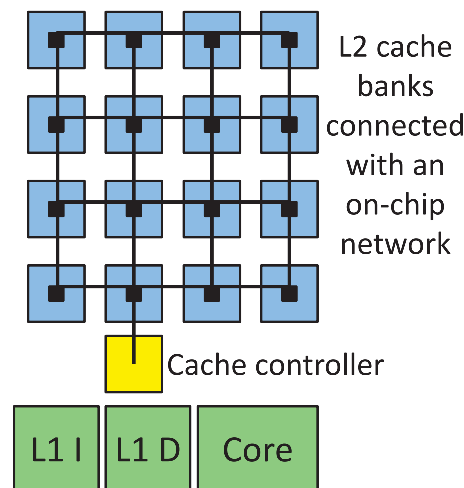


Figure 1.8: A NUCA L2 cache connected to a single core [1].

Physical Design

Kim et al. [1] consider a large L2 cache that has a single cache controller feeding one processor core (see Figure 1.8). In terms of physical layout, they consider two implementations. The first employs a dedicated channel between each of the many cache banks and the cache controller. While private channels can provide low contention and low routing overheads for each access, the high metal area requirements of these private channels are prohibitive. Such a design would also not easily scale to multiple cache controllers or cores. The second layout employs a packet-switched on-chip network with a grid topology. This ensures a tolerable metal area requirement while still providing high bandwidth and relatively low contention. Multiple cache controllers (cores) can be easily supported by linking them to routers on the periphery of the grid (or any router for that matter). While Kim et al. [1] advocate the use of *"lightweight"* routers that support 1-cycle hops, it is not yet clear if such routers can be designed while efficiently supporting the flow control needs of the cache network (more on routers in Chapter 4). If such lightweight routers exist, Kim et al. correctly point out that a highly-banked cache structure is desireable: it reduces access time within a bank, reduces contention at banks and routers, supports higher overall bandwidth, and provides finer-grain control of cache resources.

Logical Policies

Logical policies for data management must address the following three issues: (1) *Mapping:* the possible locations for a data block, (2) *Search:* the mechanisms required to locate a data block, and (3) *Movement:* the mechanisms required to change a block's location.

The simplest mapping policy distributes the sets of the cache across banks while co-locating all ways of a set in one bank. As a result, the block address and its corresponding cache index bits are enough to locate the unique bank that houses that set. The request is routed to that bank, tag comparison is performed for all ways in that set, and the appropriate data block is returned to the cache controller. Since the mapping of a data block to a bank is unique, such an architecture is known as *Static-NUCA* or *S-NUCA*. This design does not support movement of a block between banks and does not require mechanisms to search for a block.

An alternative mapping policy distributes ways and sets across banks. The *W* ways of a set can be distributed across *W* different banks. Policies must be defined to determine where a block is placed upon fetch. Similarly, policies are required to move data blocks between ways in order to minimize average access times. Because a block is allowed to move between banks, such an architecture is referred to as *Dynamic-NUCA* or *D-NUCA*. Finally, a search mechanism is required to quickly locate a block that may be in one of *W* different banks. Kim et al. consider several policies for each of these and the salient ones are described here.

The search of a block can happen in an *incremental* manner, *i.e.*, the closest or most likely bank is first looked up, and if the block is not found there, the next likely bank is looked up. Alternatively, a *multicast* search operation can be carried out where the request is sent to all candidate banks, and they are searched simultaneously. The second approach will yield higher performance (unless it introduces an inordinate amount of network contention) but also higher power. Combinations

of the two are possible where, for example, multicast happens over the most likely banks, followed by an incremental search over the remaining banks. Kim et al. propose a *Smart Search* mechanism where a partial tag (six bits) for each block is stored at the cache controller. A look-up of this partial tag structure helps identify a small subset of banks that likely have the requested data and only those banks must now be searched. While such an approach is very effective for a single-core design where all block replacements happen via the cache controller, this design does not scale as well to multi-core designs where partial tags must be redundantly maintained at potentially many cache controllers. The next chapter discusses alternative solutions proposed by Chishti et al. [23, 24]. To date, efficient search in a D-NUCA cache remains an open problem.

To allow frequently accessed blocks to migrate to banks closer to the cache controller, Kim et al. employ a *Generational Promotion* mechanism (and also consider several alternatives). On a cache miss, the fetched block is placed in a way in the furthest bank. Upon every subsequent hit, the block swaps locations with the block that resides in the adjacent bank (way) and edges closer to the cache controller.

Summary

Kim et al. show that NUCA caches are a clear winner over similar sized UCA caches and multi-level hierarchies (although, note that an N-way D-NUCA cache is similar in behavior to a non-inclusive N-level cache hierarchy). D-NUCA policies offer a performance benefit in the 10% range over S-NUCA and this benefit grows to 17% if a smart search mechanism is incorporated. This argues for the use of "clever" data placement, but the feasibility of D-NUCA mechanisms is somewhat questionable. Data movement is inherently complex; consider the example where a block is being searched for while it is in the process of migrating and the mechanisms that must be incorporated to handle such corner cases. The feasibility of smart search, especially in the multi-core domain, is a major challenge. While the performance of D-NUCA is attractive and has sparked much research, recent work shows that it may be possible to design cache architectures that combine the hardware simplicity of S-NUCA and the high performance of D-NUCA. Chapter 2 will discuss several of these related bodies of work.

1.4 INCLUSION

For much of the discussions in this book, we will assume inclusive cache hierarchies because they are easier to reason about. However, many research evaluations and commercial processors employ non-inclusive hierarchies as well. It is worth understanding the implications of this important design choice. Unfortunately, research papers (including those by the authors of this book) often neglect to mention assumptions on inclusion. Section 3.2.1 discusses a recent paper [25] that evaluates a few considerations in defining the inclusion policy.

If the L1-L2 hierarchy is inclusive, it means that every block in L1 has a back-up copy in L2. The following policy ensures inclusion: when a block is evicted from L2, its copy in the L1 cache is also evicted. If a single L2 cache is shared by multiple L1 caches, the copies in all L1s are evicted. This is an operation very similar to L1 block invalidations in a cache coherence protocol.

The primary advantage of an inclusive hierarchy in a multi-core is the ease in locating a data block upon an L1 miss – either a copy of the block will be found in L2, or the L2 will point to a modified version of the block in some L1, or an L2 miss will indicate that the request can be directly sent to the next level of the hierarchy. The L2 cache is also a central point when handling coherence requests from lower levels of the hierarchy (L3 or off-chip). The disadvantage of an inclusive hierarchy is the wasted space because most L1 blocks have redundant copies in L2.

Some L1-L2 hierarchies are designed to be exclusive (a data block will be found in either an L1 cache or the L2 cache, but not in both) or non-inclusive (there is no guarantee that an L1 block has a back-up copy in L2). Data block search is more complex in this setting: on an L1 miss, other L1 caches and the L2 will have to be looked up. If a snooping-based coherence protocol is employed between the L1s and L2, this is not a major overhead as a broadcast and search happens over all L1s on every L1 miss. This search of L1s must be done even when handling coherence requests from lower levels of the hierarchy. However, just as with snooping-based protocols, the search operation does not scale well as the number of L1 caches is increased. The advantage, of course, is the higher overall cache capacity because there is little (or no) duplication of blocks.

Another basic implementation choice is the use of write-through or write-back policies. Either is trivially compatible with an inclusive L1-L2 hierarchy, with write-through policies yielding higher performance if supported by sufficiently high interconnect bandwidth and power budget. A write-through policy ensures that shared blocks can be quickly found in the L2 cache without having to look in the L1 cache of another core. A writeback cache is typically appropriate for a non-inclusive hierarchy.

Optimal Collaborative Caching: Theory and Applications

by

Xiaoming Gu

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Chen Ding

Department of Computer Science
Arts, Sciences & Engineering
Edmund A. Hajim School of Engineering & Applied Sciences

University of Rochester
Rochester, New York

2014

Biographical Sketch

Xiaoming Gu was born in Handan, Hebei, China, in 1980. He started his study in computer science at the University of Science and Technology of China in 1998 and received his B.S. degree there in 2003. In 2006, he received his first M.S. degree at the Institute of Computing Technology, Chinese Academy of Sciences. In 2008, he received his second M.S. degree at the University of Rochester. After one-year of work at Intel Beijing, he continued his Ph.D. study at the University of Rochester in 2009 under the direction of Professor Chen Ding. He did a summer internship at AMD in 2011. Since the summer of 2012, he has been a full-time software engineer at Azul Systems.

Much of this dissertation is from the following four workshops and conference papers on optimal collaborative caching:

- *P-OPT: Program-directed Optimal Cache Management*, Xiaoming Gu, Tongxin Bai, Yaoqing Gao, Chengliang Zhang, Roch Archambault, and Chen Ding, 21st Workshop on Languages and Compilers for Parallel Computing (LCPC'08)

- *On the Theory and Potential of LRU-MRU Collaborative Cache Management*, Xiaoming Gu and Chen Ding, 10th International Symposium on Memory Management (ISMM'11)

- *A Generalized Theory of Collaborative Caching*, Xiaoming Gu and Chen Ding, 11th International Symposium on Memory Management (ISMM'12)

- *Pacman: Program-Assisted Cache Management*, Jacob Brock, Xiaoming Gu, Bin Bao, and Chen Ding, 12th International Symposium on Memory Management (ISMM'13)

Other than the findings described in this dissertation, the author has studied several other areas:

- Reuse distance distribution for random access

  Reuse Distance Distribution in Random Access (position abstract), Xiaoming Gu and Chen Ding, 5th Workshop on Memory Systems Performance and Correctness (MSPC'08)

- Spatial locality modeling for program tuning

  A Component Model of Spatial Locality, Xiaoming Gu, Ian Christopher, Tongxin Bai, Chengliang Zhang, and Chen Ding, 8th International Symposium on Memory Management (ISMM'09)

- Software parallelization by hint

  Continuous Speculative Program Parallelization in Software (poster paper), Chao Zhang, Chen Ding, Xiaoming Gu, Kirk Kelsey, Tongxin Bai, and Chen Ding, 15th Symposium on Principles and Practice of Parallel Programming (PPoPP'10)

- Resource-based memory management

  Waste Not, Want not: Resource-based Garbage Collection in a Shared Environment, Matthew Hertz, Stephen Kane, Elizabeth Keudel, Tongxin Bai, Chen Ding, Xiaoming Gu, and Jonathan Bard, 10th International Symposium on Memory Management (ISMM'11)

Acknowledgments

First of all, I would like to sincerely thank my advisor, Professor Chen Ding. Without him, I cannot imagine that I would have been able to finish my Ph.D. study. He is a role model for me in many ways.

Professor Hertz, Professor Ipek, and Professor Scott have sat patiently on my thesis committee for years. I appreciate their insightful suggestions and invaluable time investment very much.

I received help from all the people in the department. They made my study much smoother. In particular, I would like to thank the students in the compiler group: Chengliang Zhang, Kirk Kelsey, Tongxin Bai, Bin Bao, Xiaoya Xiang, Hao Luo, and Jacob Brock. In this dissertation, Jacob Brock made a significant contribution to the loop splitting part, and Bin Bao performed the tests on real hardware. I greatly appreciate their invaluable efforts. I also received considerable help from Hao Zhang, Arrvindh Shriraman, Xiao Zhang, Qi Ge, Daphne Liu, Amal Fahad, Licheng Fang, Zhuan Chen, Li Lu, and Marzieh Bazrafshan.

Most of this dissertation comes from my previous publications. I would like to thank Jorge Albericio, Roch Archambault, Luke K. Dalessandro, Sandhya Dwarkadas, Yaoqing Gao, Li Shen, Xipeng Shen, Lingxiang Xiang, and the anonymous reviewers at LCPC'08, ISMM'11, ISMM'12, and ISMM'13. Special appreciation goes to Kathryn McKinley, who suggested the idea of priority hints.

I would also like to give my sincerest thanks to my colleagues at Azul Systems. They helped me a lot in my full-time job. I cannot imagine that I would have had

time for my dissertation without their help. I really enjoy working with them, especially my manager Bean Anderson. I am proud of working with these great people.

This dissertation is dedicated to my parents and sister. I offer them my thanks for their unconditional love, which kept me warm. My final thanks belong to my girlfriend Zhou Yi, who brings a great deal of fun into my life.

Abstract

On most modern computers, most memory accesses happen in cache, and its usage increasingly affects the performance, stability, and energy consumption of the whole system. The traditional solution divides the memory problem and conquers the pieces separately: software to improve cache locality and hardware to improve cache speed and capacity. A recent approach called collaborative caching breaks the rigid division by allowing software to provide cache hints to influence hardware cache management. In traditional caching, the hardware infers the "importance" of data and manages cache based on the inferred priority. In collaborative caching, software specifies the "importance" of data and changes the priority in which the hardware manages the data.

The dissertation presents the theory and applications of optimal collaborative caching. Through software-hardware collaboration, the goal is not to improve existing solutions but to obtain the optimal solution. Toward this goal, I tackle both theoretical and practical issues. I show in theory that efficient hardware similar to the existing cache is sufficient to produce optimal results. Based on the theory, I describe practical techniques on the software side and evaluate the combined solution on both simulated and real hardware.

Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Professors Chen Ding, Engin Ipek, and Michael Scott of the Department of Computer Science at the University of Rochester and Professor Matthew Hertz of the Department of Computer Science at Canisius College. Section 4.3 was conducted in part by Jacob Brock and Bin Bao. All other work conducted for the dissertation was completed by Xiaoming Gu independently. Graduate study was supported by NSF awards CNS-0834566 and CCF-0963759, and by an IBM fellowship #TOR08003-12.

Table of Contents

Biographical Sketch **ii**

Acknowledgments **iv**

Abstract **vi**

Contributors and Funding Sources **vii**

List of Figures **xii**

List of Tables **xvi**

1 Introduction **1**

1.1 The Thesis 1

1.2 Memory Hierarchy and Cache 2

1.3 From LRU to Optimal 4

1.3.1 Previous Solutions 5

1.3.2 From OPT Cache to Collaborative Cache 6

1.3.3 The Belady Anomaly 8

1.3.4 From One-Size to All-Size Optimization 8

1.4 Contributions and Organization 10

2 Related Work **12**

2.1 Collaborative Cache Hardware 12

2.2 Collaborative Caching Software 13

2.2.1 Hint Insertion 13

2.2.2 Cache Partitioning 14

2.3 Non-collaborative Solutions 15

2.3.1 Inclusion Property and Stack Distance 15

2.3.2 Program Analysis and Optimization 15

2.3.3 Non-LRU Cache 16

2.3.4 Memory Management 17

2.4 Optimal Caching 17

3 Theoretical Properties of Collaborative Caching **19**

3.1 Introduction 19

3.2 Background on Non-collaborative Caching 22

3.3 LRU-MRU Cache 26

3.3.1 Cache Design 26

3.3.2 Optimal LRU-MRU Hints 28

3.3.3 Optimality 30

3.3.4 Multi-size Optimality 33

3.3.5 The OPT* Algorithm 35

3.3.6 Inclusion Property 36

3.3.7 LRU-MRU Stack Distance 38

3.4 Trespass LRU Cache 46

3.4.1 Optimality 47

3.4.2 Multi-size Optimality 50
3.5 Priority LRU Cache 52
3.5.1 Inclusion Property 56
3.5.2 Non-uniform Inclusion 62
3.5.3 Priority LRU Stack Distance 64
3.5.4 Optimality of Priority Hint for All Cache Sizes 74
3.6 Inclusive Cache Hierarchy 75
3.7 Summary 77

4 Pacman: Program-assisted Cache Management 78

4.1 Introduction 78
4.2 Reference-based Pacman 81
4.2.1 The Design 81
4.2.2 Experiment Setup 82
4.2.3 The LRU-OPT Gap 84
4.2.4 The Effect of Reference Hints 85
4.2.5 The Effect of Program Input 87
4.2.6 The Impact of the MRU Ratio Threshold 89
4.2.7 A Closer Look at SOR 90
4.2.8 The Miss Ratio Curves of LRU, OPT, and Pacman 95
4.3 Loop-based Pacman 97
4.3.1 Forward OPT Distance Profiling 98
4.3.2 Pattern Recognition 99
4.3.3 Loop Splitting and Hint Insertion 103
4.3.4 Cross-Input Pattern Prediction 105

4.3.5 Evaluation Setup 107

4.3.6 Optimal Caching of Group Spatial Reuse in SOR 107

4.3.7 Program-directed Cache Allocation in Swim 111

4.3.8 Six Other Programs 113

4.3.9 Comparison to Dynamic Insertion Policy 117

4.3.10 Pacman on Real Hardware 119

4.4 Summary 122

5 Conclusions and Future Work **124**

Bibliography **127**

List of Figures

1.1 LRU & OPT miss ratios of a streaming application on power of 2 cache sizes from 64KB to 32MB. OPT may divide a large working set for caching but LRU does not. 4

3.1 Normal LRU at a miss: w is placed at the top of the stack, evicting S_m. 26

3.2 Normal LRU at hit: w, assuming at entry S_3, is moved to the top of the stack. 26

3.3 Bypass MRU at a miss: the bypass posits w at the bottom of the stack, evicting S_m. 27

3.4 Bypass MRU at a hit: the bypass moves $S_3(w)$ to the bottom of the stack. 27

3.5 An example of optimal LRU-MRU hint insertion. For each access, the forward OPT distance is the OPT distance for the following access to the same data element, and the optimal hint for each access is LRU if and only if the forward OPT distance is equal to or less than the cache size. 29

3.6 Comparing LRU, MRU, OPT, and LRU-MRU. LRU-MRU is optimal for the targeted cache size. 30

3.7 a_i is selected for bypass for a given cache size during an OPT cache simulation. 31

3.8 Trespass MRU at a miss: the trespass posits w at the top of the stack, evicting S_1 47

3.9 Trespass MRU at a hit: the trespass raises $S_3(w)$ to the top of the stack, evicting S_1 47

3.10 Element a_i is selected for trespass for a given cache size during an OPT cache simulation 47

3.11 Two cases of data hit in the priority cache when the data block w, at position j in cache, is accessed with a priority i. 54

3.12 Two cases of data hit in the priority cache when the data block w, at position j in cache, is accessed with a priority i. 55

3.13 Two cases of data miss in the priority cache when the data block w, not in cache before the access, is accessed with priority i. . . . 56

3.14 An example of non-uniform inclusion. The Priority LRU observes the inclusion principle but permits data to locate in different positions in the smaller cache than in the larger cache. In this example, after time 8, A locates at a lower position in the size-5 cache than in the size-6 cache. 63

3.15 For the same trace in Figure 3.14, the access at time 9 is a miss in the size-4 cache. 65

3.16 An example of Priority LRU stack simulation. The trace has nine accesses to four data elements. A data element may locate at different stack positions depending on cache sizes. All possible positions for each data element are tracked by its *span* list, shown in each row. Cache sizes are shown by the header row. 66

3.17 The following five steps of the example of Priority LRU stack simulation appear in Figure 3.16. 67

3.18 The Inclusive Cache Hierarchy: Inclusive caches are organized in a hierarchy based on the "implemented-by" relationship. Limited collaborative caching of LRU-MRU in Section 3.3 subsumes non-collaborative schemes of LRU, MRU and OPT [41]. Priority LRU subsumes LRU-MRU and other prior collaborative caches. 76

4.1 An example of reference-based pacman hint insertion. 81

4.2 The miss curves of 189.lucas on fully associative caches 86

4.3 The miss curves of 434.zeusmp on fully associative caches 87

4.4 The miss curves of 171.swim on two different inputs. The curves have an identical shape but cover different cache-size ranges: between 1KB and 4MB in the upper graph and between 1KB and 16MB in the lower graph. 89

4.5 The impact of the MRU ratio threshold for 173.applu at 512KB . 90

4.6 The miss curves of 173.applu on fully associative caches 91

4.7 The SOR kernel computation 91

4.8 The gap between LRU and OPT in SOR 92

4.9 The SOR kernel loop in SSA form with Pacman transformation. `M = N = 512` and `NUM_STEPS = 10`. 93

4.10 The MRU ratio curves of SOR on fully associative caches with cache line size 64B 94

4.11 The miss curves of SOR on fully associative caches with cache line size 64B 95

4.12 172.mgrid 96

4.13 183.equake 96

4.14 410.bwaves 96

4.15 433.milc 97

4.16 437.leslie3d 97

4.17 An example of loop-based Pacman hint insertion. 98

4.18 The OPT distances exhibited by a reference in an execution of *Swim*. 101

4.19 The percent reduction in cache misses of *SOR* by Pacman and OPT over LRU 110

4.20 Pacman is tested on *swim* when the input size (b), array shape (c) and both (d) change. The executions in (a,b) use the same loop splitting points, so do the executions (c,d). 112

4.21 The improvements by Pacman and OPT over LRU 114

4.22 The improvements by Pacman and OPT over LRU 115

4.23 One of the representative OPT distance patterns in *applu*. The two graphs show the same reference with the same series of the OPT distances as the y coordinate but with different x coordinates as the iteration count in two of the five enclosing loops. 116

4.24 An OpenMP example: the inner loop updates the array element by element; the outer loop corresponds to the time step. 119

4.25 The performance comparison on Intel Xeon E5520 with hardware prefetching 120

4.26 The performance comparison on Intel Xeon E5520 without hardware prefetching 121

List of Tables

3.1 An example for LRU cache 23

3.2 An example for MRU cache 25

3.3 An example for OPT cache 25

3.4 An example showing optimal LRU-MRU with cache size 2 31

3.5 An example of LRU-MRU with cache size 3. The memory trace is the same one used in Table 3.4. The two examples together show that multi-size optimal LRU-MRU does not have the inclusion property. 34

3.6 Example one-pass simulation of LRU-MRU cache 42

3.7 Two examples showing Trespass LRU can be optimal and multi-size optimal Trespass LRU holds inclusion property (unlike multi-size optimal LRU-MRU) 48

3.8 The nine cases for the next access x_{t+1} to d' with a priority p'. . . 57

3.9 In the six subcases of Case I in Table 3.8, the access x_{t+1} is a hit in both C_1 and C_2. A hit can be one of the cases shown in Figure 3.11 and 3.12 except the bypass case. 57

3.10 In the three subcases of case II in Table 3.8, the access x_{t+1} misses in C_1 but hits in C_2. The hit and miss cases are shown in Figures 3.11, 3.12 and 3.13. 59

3.11 The three subcases of x_{t+1} of case V 60

3.12 The measured overhead of Algorithm 3.2 when computing the Priority LRU stack distance over a random-access trace with 10 million accesses to 1024 data elements with random priorities. The maximal priority number ranges from 1 to 1 million. The space is measured by the number of being stored spans. The time is measured by the number of calls to a span update. In most columns, the time and space costs are close to LRU stack simulation. The highest cost is incurred when the priority is up to 1024, but the worst cost is still far smaller than the theoretical upper bound. . . 73

4.1 The 10 test programs 83

4.2 The LRU-OPT gap and the Pacman improvement. The average improvement is the arithmetic mean of the improvement for all cache sizes between 1KB and data size. 85

4.3 The statistics of the 8 workloads 108

4.4 Pacman makes full utilization of cache space and gradually reduces the miss ratio as the cache size increases 110

1 Introduction

1.1 The Thesis

Since the late 1990s, an increasing number of hardware systems have been built or proposed to provide a cache hint interface for software to influence cache management. Examples include cache hints on Intel Itanium [11], bypassing and non-temporal accesses on IBM Power series [50], evict-me bit [57], and non-temporal stores on Intel x86 processors. Wang et al. called a combined software-hardware solution *collaborative caching* [57].

As a form of memory, cache only sees the past and present data accesses. The traditional, non-collaborative management is inherently limited by what it has seen. Collaborative caching allows a program to provide hints about its data usage so that hardware can improve the cache utilization by keeping the most active program data in cache. It exploits the synergy between the hardware considering past behavior and the software supplying the information about the future.

The techniques of collaborative caching were pioneered by a pair of studies. Wang et al. used the dependence information to label "evict-me" bits for data sets too large to fit in cache [57]. Beyls and D'Hollander placed data into different

levels of the cache based on their reuse distances [11]. In these and many other studies, the goal is to improve the existing cache management. The effect of past work depends on specific designs and design parameters. It is unclear what the ultimate potential of collaborative caching is. While significant improvements have been made, it is not clear how much of potential remains.

It is the author's belief that collaborative caching holds the key to solving the long-standing problem of optimal cache management. Toward this goal, this dissertation studies optimal collaborative caching.

Thesis Statement. *Optimal cache management can be done efficiently through software-hardware collaboration. Collaborative hardware can be as simple and efficient as existing cache and as robust against incorrect software control. Collaborative software can be general and optimized for all cache sizes.*

1.2 Memory Hierarchy and Cache

In computing, there is a fundamental conflict between computing speed and data capacity.[1] A signal cannot travel faster than the speed of light; consequently, the faster the data access is, the smaller the amount of data it can reach. The solution is to trade space for speed. The data is stored in a memory hierarchy. At each level, the capacity increases while the access speed decreases.

There are two other ways for modern computers to trade space for speed. Firstly, computer cache uses a special structure called the SRAM, which is ten times faster than DRAM, the structure used for main memory, at the cost of storing only one fourth or one sixth as much data. Secondly, to store data more densely, it is not a good idea to run a wire to each memory or cache cell, so the bandwidth of data access is limited in exchange for a higher data capacity.

[1] The characterization in the first two paragraphs is based primarily on Peter Sanders in Section 1.1 in [43].

Memory hierarchy provides fast access to active data. Data is active if it is being used by a running program. Denning defined the set of data that a program uses frequently as the working set [18, 19]. Depending on different levels of frequencies, there are multiple levels or multiple working sets.

It is an unavoidable problem for any computing system to find ways to manage active data in a limited space. There are two basic sub-problems: data placement and data replacement. When a piece of data is needed, it has to be brought in. If the available space is full, some resident data must be selected and evicted.

Computer cache is a level of memory hierarchy where hardware manages the data. It improves programmability because the decisions of data placement and replacement are done in hardware. Software sees only a uniform memory. There is not need to change a program when the program is run on a machine with a larger cache.

The cache interface represents a fundamental division of labor: the consumption of data in software and the management by hardware. To remove the inter-dependence, the division is "air tight"; that is, software has no direct control over data management. So far, the devision is effective and pervasive. The problems of cache implementation are well encapsulated, and computer industry has found highly optimized solutions.

The dominant solution is based on the management policy called Least Recently Used (LRU). At eviction, the data that is least recently used (remains unused the longest) is replaced. On the software side, LRU keeps the most recently used data and favors the working set of a program. It supports programs with good *locality*. The currently used data or their neighbors are likely reused in the near future. On the hardware side, LRU is simple and can be implemented efficiently. Set associativity has a fraction of the cost of the full LRU but can have the same effect once the associativity is reasonably high [12]. Pseudo-LRU takes only $N - 1$ bits per N-way cache set instead of $log_2 N!$ bits in LRU [53].

Research on collaborative caching breaks through this long accepted limitation and opens new channels of control for software over hardware. Before making the move, we must have compelling reasons and must have a good understanding of the difficulties that are involved.

1.3 From LRU to Optimal

LRU has a well-known thrashing problem. If the working set of a program is too large to cache, no cache can provide reuse for the whole data. The problem with LRU is that it may not provide reuse for any data.

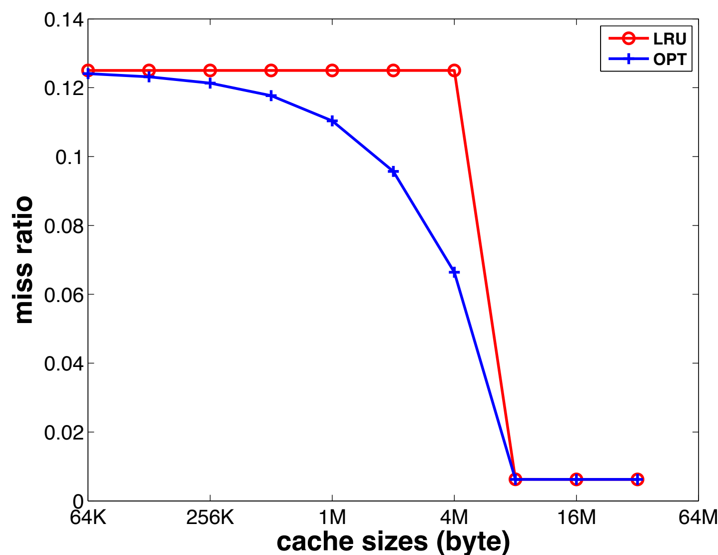


Figure 1.1: LRU & OPT miss ratios of a streaming application on power of 2 cache sizes from 64KB to 32MB. OPT may divide a large working set for caching but LRU does not.

Consider a streaming program that repeatedly traverses a 5MB array 1000 times. Figure 1.1 shows the miss ratio curves for cache sizes at all power-of-two numbers between 64KB and 32MB. The first is LRU cache, 16-way set associative with 64-byte cache blocks. When the cache size is smaller than 5MB, LRU has no effective cache reuse. It has the same high miss ratio at the 4MB size as it is at

the 64KB size. The reason is thrashing. The previous data is constantly evicted by the new data. Only when the cache size exceeds 5MB does the miss ratio drop to zero. The sharp knee of the miss ratio curve shows the size of the working set.

A better solution is to keep a portion of the working set in cache. If the cache size is 4MB, we can store 4MB of the 5MB in cache and leave the remaining 1MB outside. This cannot be done by LRU, which has to store the 1MB portion when it is accessed. The second curve shows another replacement policy called OPT, which stores a part of the working set that fits in cache. The miss ratio drops proportionally with the increase in cache size.

The example shows that software may better manage cache memory by targeting specific parts of working sets. Despite the theorized performance bounds between LRU and optimal, this is an example showing that LRU can be arbitrarily inferior to what is obtainable by incorporating software knowledge.

1.3.1 Previous Solutions

The thrashing problem in LRU has been extensively studied. Well-established solutions in program optimization include cache blocking, loop fusion, many other loop nest transformations and structure splitting, and many other cache conscious data and code layout techniques [6,21,70]. In hardware cache design, the solutions include dead-block predictor, forward time distance predictor, adaptive cache insertion, less reuse filter, virtual victim cache, and globalized placement.

While compiler optimization is purely software, the non-LRU cache design is purely hardware. A software solution can change the data usage but not the management policy in hardware. A hardware solution can change the management policy but not the data usage pattern in software. They are fundamentally limited by the rigid software-hardware border in the traditional cache interface.

An increasing number of modern machines have augmented the traditional cache interface to permit software control. Examples include the non-temporal MOVNTQ SSE instruction on x86 in 1999, the MOVNTPD instruction for non-temporal writes in SSE2 in 2001, the placement hints on Intel Itanium processors since 2001, and bypass memory instructions on IBM machines since Power 5 in 2005. These instructions are called *cache hints*.

A number of techniques have been developed using these hints. They include the pioneering work in collaborative caching as a compiler optimization in 2002 [10, 11, 57] and more recently specialized uses in re-initializing memory in a garbage collector [66] and in string processing [47] in 2011. These studies had the goal of improving the LRU management with program knowledge and were effective in doing so. However, the significance is more profound than the practical improvements. By implicating software, cache management is no longer limited by the traditional software-hardware division. Inspired by and complementary to these earlier studies, this dissertation explores the limit of this synergy—the range of possibilities that arise when combining software flexibility and hardware efficiency.

1.3.2 From OPT Cache to Collaborative Cache

This work aims not to improve LRU but to optimize cache management. The goal is not a better solution but the best possible solution. There are challenges in theory and in practice.

First, we need a theory of optimal collaborative caching. The past work makes incremental improvement over LRU. The claim of improvement can be demonstrated by comparing these improvements against one or more baseline designs. The claim of optimality, however, must be accompanied by a proof that no other cache management can do better.

Secondly, for a theory to matter in practice, there must be a way to realize it. Consider the OPT replacement policy, which is to *replace the data whose next use is the furthest in time*. OPT is known to be optimal since 1960s. No OPT cache has ever been built nor will it be because OPT is impractical.

There are three main obstacles for hardware OPT. First, OPT requires clairvoyance. It needs to know the future of data accesses. Second, assuming clairvoyance, OPT is time and space inefficient to implement. Each data item has to remember its next access time, and each access triggers a comparison of access times. In comparison, LRU wastes no space in storing any time tags and naturally no operations in comparing them. Considering how cache must be optimized for space and speed, it is questionable whether we can add these overheads and still produce performance competitive to the ultra-optimized and dense LRU. Third, beside the complexity of management inside the cache, another problem is the cost of communication between software and hardware, if we use collaborative caching. Conveying the access time requires the encoding it and adding extra bits to every memory access.

Now we consider the three problems of OPT cache can be solved by collaborative cache. In collaborative caching, the burden of clairvoyance is moved to software. The problem is actually easier. While a hardware solution would have to look through possibly billions of data accesses to find the next reuse, a compiler can derive the information from the program code. If the code is complex and not amenable to program analysis, program behavior analysis can infer the data usage patterns in large executions by analyzing the traces of smaller training runs.

The next two problems are the internal complexity and the interface complexity of the collaborative cache. We will show that collaborative cache can use simple variations of LRU in the place of OPT, using the software hints to eliminate the need for storing access times or performing comparison logic. The cache interface can use just one extra bit for each memory access. Chapter 3 gives the

formal proofs of the optimality of the collaborative cache.

1.3.3 The Belady Anomaly

The Belady anomaly happens if a larger cache incurs more misses than a smaller cache does [7]. It is easy to see that optimal cache management rules out the Belady anomaly. A larger cache can at least be used as a smaller cache, so the best solution is at least as good as before.

We will review the background of traditional, non-collaborative cache in Section 3.2 and show that LRU does not have this anomaly either. When there is more cache space, the performance of LRU cache increases or stays the same but does not decrease.

In collaborative cache, the Belady anomaly would mean that the collaboration on one cache size becomes counter productive in other sizes in that the collaborative cache cannot utilize the additional space. Worse, the addition of space causes it to manage the existing space less efficiently.

Intuitively, the anomaly should not happen. In LRU, the hardware infers the "importance" of data and manages cache based on the inferred priority. In collaborative cache, software specifies the "importance" of data and changes the priority in which the hardware manages the data. As the cache size increases, the data important at the smaller cache remains important. The collaborative cache should be able to cache them at least as well. Indeed, Chapter 3 will prove that this is indeed the case for the majority of collaborative cache designs, where the Belady anomaly cannot happen.

1.3.4 From One-Size to All-Size Optimization

An inherent limitation in optimal collaborative cache is the size dependence of the optimization. For efficiency, we insist on a single bit per access in communicat-

ing between software and hardware. As a result, the collaborative cache cannot optimize for all cache sizes based on this single bit of information. The same hinted execution may be the best one-size solution. Ideally, we want a program be optimized by an all-size solution.

Section 3.5.4 gives a solution where hardware transforms multi-bit priority hints into a single-bit cache hints and enables the use of the same hints to optimize for all cache sizes. Still, it is desirable to solve the all-size optimization problem using only single-bit hints between software and hardware. Chapter 4 presents a practical implementation called Pacman to solve this problem.

Pacman uses off-line profiling analysis. It analyzes the patten of optimal cache management for all cache sizes. Chapter 4 describes two techniques. The first chooses the hint for each memory reference. It is generally applicable but treats all accesses of the same reference in the same way. The second technique targets loops. It transforms a loop to create differential treatment of its memory accesses based on the iteration counter. The loop-based Pacman automatically changes the software hints for different size caches.

There are many factors affecting a practical implementation. Cache blocks contain multiple data items. Loop analysis must consider both spatial and temporal data reuse in cache management. The cache is set associative, not fully associative. Even within a single set, modern cache is not exactly LRU. We have to evaluate through experimentation. The experiments should compare cache performance for not just a single cache size but as many sizes as possible. The memory usage of a program changes with its input. Pacman should optimize for different inputs or at least provide a robust performance across all inputs. Finally, better caching does not necessarily mean faster running speed. Performance testing on a real machine will consider all factors, including the cost incurred when using special memory instructions. Chapter 4 will address these issues.

1.4 Contributions and Organization

This dissertation makes the following contributions:

1. It lays down the theoretical foundation for collaborative caching:
   - Optimality—LRU-MRU caching can be optimal, and the optimal cache hints can be independent of cache size.
   - LRU-MRU stack distance—LRU-MRU caching is proved as a stack algorithm holding inclusion property. A new stack distance algorithm is devised for it.
   - Generalization—Cache hints are extended to a generalized form from 1-bit for LRU-MRU. Priority LRU, a new stack algorithm based on the generalized priority hint, is discovered holding inclusion property but in a nonuniform way. A new stack distance algorithm is devised for Priority LRU. Generalized cache hints can be used to achieve optimal caching for all cache sizes.
2. It explores the applications of collaborative caching:
   - Reference hints—A heuristic-based solution called Pacman is presented to use optimal trace-level hints to decide program-level reference hints in a binary executable.
   - Refinement by loop splitting—A refined Pacman with loop splitting is proposed to separate run-time accesses with different hints at program level. The cache-size dependent limitation for optimal caching is also removed by using priority hint.

This dissertation is organized as follows. Chapter 2 presents related work. Chapter 3 shows in theory how collaborative caching can use efficient hardware

and still be able to optimize cache management. Chapter 4 presents the Pacman system, which enables a program to determine, efficiently at run time for each memory access, whether the accessed data should be cached or not and to efficiently communicate this decision to hardware at run time at each access. The hints are made robust across input and cache sizes, and there is no trace analysis or code generation at run time. The last chapter is the conclusions and future work.

2 Related Work

2.1 Collaborative Cache Hardware

The ISA of Intel Itanium extends the interface of the memory instruction to provide source and target hints [5]. The source hint suggests where data is expected, and the target hint suggests which level cache the data should be kept. The target hint changes the cache replacement decisions in hardware. IBM Power processors support bypass memory accesses that do not keep the accessed data in cache [50]. Wang et al. proposed an interface to tag cache data with evict-me bits [57]. Another way for software control is cache partitioning. Ding et al. developed a system called ULCC (User Level Cache Control), which uses virtual-to-physical page mapping to partition the cache to separately store high locality and low locality data [23].

The goal in these studies was to allow software control to improve cache management. They do not explore the limit of collaborative caching, i.e. whether the cache can be optimal. This dissertation will show what type of collaborative cache can attain optimal management.

2.2 Collaborative Caching Software

2.2.1 Hint Insertion

Collaborative caching was pioneered by Wang et al. [57] and Beyls and D'Hollander [10, 11]. The studies were based on a common idea, which is to evict data whose forward reuse distance is larger than the cache size. Wang et al. used compiler analysis to identify self and group reuse in loops [42, 57, 58] and select array references to tag with the evict-me bit. They showed that collaborative caching can be combined with prefetching to further improve performance.

Beyls and D'Hollander used profiling analysis to measure the reuse distance distribution for each program reference. They added cache hint specifiers on Intel Itanium and improved average performance by 10% for scientific code and 4% for integer code [10]. Profiling analysis is input specific. Fang et al. showed a technique that accurately predicts how the reuse distances of a memory reference change across inputs [26]. Beyls and D'Hollander later developed a static analysis called reuse-distance equations and obtained similar improvements without profiling [11]. Compiler analysis of reuse distance was also studied by Cascaval and Padua for scientific code [14] and Chauhan and Shei for Matlab programs [15].

The prior methods used reuse distance to identify data in small-size working sets for caching. It was unclear whether and how much cache utilization could be further improved. The goal of this work is optimal collaborative caching. The practical solution, Pacman, uses the OPT distance instead of the reuse distance for caching analysis. In the case when a larger working set is too large, the accesses have the same reuse distance but different OPT distances. By analyzing the difference, Pacman can partition the large working set, choose a partial set to cache and hence utilize the available cache space fully.

Two recent papers show the benefits of collaborative caching on current x86

processors. Yang et al. used non-temporal writes for zero initialization in JVM to reduce cache pollution [66]. Rus et al. used non-temporal prefetches and writes to specialize string operations like `memcpy()`, based on the data reuse information in certain static program contexts [47]. They showed that significant improvements are already possible by exploiting collaborative caching on current hardware.

2.2.2 Cache Partitioning

Cache partitioning as done by ULCC provides practical improvements through software-hardware collaboration [23, 39]. It does not need access hints. Hence there is no additional overhead in memory access or cache management. When we used ULCC to cache 5MB working set in 4MB cache (by assigning the first 3.5MB to use 3.8MB cache and the other 1.5MB to just 128KB cache), we observed a 37% reduction in the execution time. Despite of the large improvement in this case, cache partitioning may not obtain optimal cache management in general. OPT, for example, does not partition the cache for exclusive use. The full cache space is available to all data at all times. Based on the OPT distance, Pacman may choose to cache a piece of data at one time and then choose to keep the same data out of the cache at another time.

Cache partitioning is data based. Collaborative caching is access based. As an allocation scheme, the latter is a form of prioritization rather than partitioning. Rather than allocating cache explicitly between data, Pacman designates some data to be of higher priority than other data. Taking an analogy in operating systems, Pacman is more like CPU scheduling than virtual memory management. When the priority is wrong, the cache space is still utilized. If we partition the cache, an incorrect partition can lead to unutilized space.

2.3 Non-collaborative Solutions

2.3.1 Inclusion Property and Stack Distance

Mattson et al. established the inclusion property and the metrics of stack distance [41]. The miss ratio of inclusive cache is monotonically non-increasing as the cache gets larger (whereas the Belady anomaly [8], more misses in larger cache, is impossible). Stack distance can be used to compute the miss ratio for cache of all sizes. They presented a collection of algorithms based on a priority list. The LRU stack distance, i.e. reuse distance in short, can be computed asymptotically faster (in near linear time for a guaranteed precision) using a (compression) tree [70]. The cost can be further reduced by statistical modeling [24, 25, 35, 49], sampling [9,13,48,55,69], footprint-based conversion [20,64], and parallelization on MPI [44] and GPU [16, 28]. Recent work has adapted the reuse distance analysis to model the locality in multi-threaded programs [22, 35, 48, 60–62].

Reuse distance and other stack distances cannot be used to analyze the collaborative cache. In fact, it was unknown before this work whether some form of stack distance exists for collaborative cache. This dissertation will prove the inclusion property for two types of collaborative cache and give the algorithms to measure their stack distances. Two new stack distances will be studied. In addition, the thesis will show a new type of inclusion property that gives rise to a new category of caching algorithms.

2.3.2 Program Analysis and Optimization

Much research has been done on improving program locality. Locality was initially defined qualitatively. The textbook concepts of temporal and spatial locality refer to the tendency for the currently accessed data or its neighbors to be accessed in the near future. To measure locality quantitatively, we often use the reuse

distance. A data access is a capacity miss if its reuse distance is greater than the cache size, so the distribution of reuse distances gives the probability of a miss at each access, i.e. the miss ratio, for a fully associative LRU cache.

Locality depends on cache management. Reuse distance assumes that the cache is managed by replacing the least recently used data (LRU). Under this policy, every data element loaded between a pair of data reuses would stay in the cache if the reused data element does. The intermediate data all contribute to the reuse distance. Under optimal cache management (OPT), the intermediate data may or may not be kept in the cache, so the OPT locality can improve over the LRU locality.

OPT locality has often been used to study the potential of caching. In this work, we use OPT distance to direct cache management — to choose which data to cache at what time and to allocate the available cache among different data. With collaborative caching, a program can still have good (OPT) locality even it cannot have good LRU locality (because of the dependence or other difficulties).

2.3.3 Non-LRU Cache

The idea of evicting dead data or least reused data early has been extensively studied in hardware cache design, including dead block predictor [38], forward time distance predictor [27], adaptive cache insertion [46], less reuse filter [63], virtual victim cache [37], and globalized placement [67]. These techniques do not require program changes but they could only collect program information by passive observation. Hardware cache by nature only sees the past and present data access and is inherently limited by what it has seen. Like non-LRU cache, collaborative cache uses non-LRU data management. However, the control is by software. Previous work was aimed to improve the cache not optimize it. For example, Qureshi et al. described a “dueling” strategy to choose the better policy

between MRU and LRU. This work aims at optimal caching, which is to find the best policy and must require software collaboration.

2.3.4 Memory Management

Garbage collectors may benefit from the knowledge of application working set size and the affinity between memory objects. Reuse distance has been used by virtual machine systems to estimate the working set size [65] and to group simultaneously used objects [68]. There have been much research in operating systems to improve beyond LRU. A number of techniques used last reuse distance instead of last access time in virtual memory management [34, 52, 71] and file caching [33]. If collaborative caching is effective for hardware cache, similar solutions may help to improve memory management as well.

2.4 Optimal Caching

The problem of optimal caching has been studied mainly in three forms. The first is the hardware question of *optimal cache management*: For a given program execution, i.e. a fixed instruction sequence and data layout, how to minimize the number of misses. Here the program is fixed, and the cache is to be optimized. If we fix the cache, we have the software question of *program optimization*. Given a given cache memory, i.e. a fixed management scheme, how to reorganize a program to minimize the number of misses. In the second problem, the cache is fixed, the program, its computation and data layout, is to be optimized. If we consider both optimization in software organization and hardware management, the ultimate problem is known as *minimal I/O complexity*. Given a computation with data D stored on disk and loaded into memory of size M ($M < D$) when being computed, how to minimize the number of communication between the memory and the disk.

The three problems are increasingly difficult. Cache management is the only one that is known to be solvable in polynomial time. Before this work, the optimal solution is theoretical. The optimal cache would need the future information that the hardware does not have and require complex and costly operations that the hardware cannot afford to do. This dissertation shows how to solve these two problems using optimal collaborative caching. We will show how software can work with simple hardware to obtain the optimal goal in theory and approach this goal in practice.

Program optimization is more complex because of the dependences that must be preserved in a program execution. Kennedy and McKinley [36] and Ding and Kennedy [21] showed that optimal loop fusion is NP hard. Surprisingly for data layout where there is no constraint on data ordering, Petrank and Rawitz showed that given the order of data access and cache management, the problem of optimal data layout is intractable unless P=NP [45]. From these results, it is easy to deduce that the I/O complexity problem is NP-hard. The I/O lower-bound has been solved for specific problems by Hong and Kung for matrix-vector and matrix-matrix multiplication, FFT, and odd-even transposition sort [31] and by Vitter for permutation [56]. An extensive overview of the algorithmic issues and solutions can be found in Meyer et al. [43] Given the intractability of program optimization and I/O minimization, optimal cache management is the best possible in theory. This dissertation will establish the theory and explore its applications.

3 Theoretical Properties of Collaborative Caching

This chapter presents three types of collaborative cache and establishes three theoretical properties: optimality, inclusion property, and generality.

3.1 Introduction

In cache as well as memory management, the least recently used replacement policy (LRU) is a common starting point. In theory, Sleator and Tarjan showed that LRU is within a bounded factor of optimal [51]. In practice, all modern microprocessors use some variation of LRU.

For collaborative cache, we use LRU as the basis. To be able to revise the default LRU policy, we add non-LRU access to the cache interface. A program can then use a mix of non-LRU accesses and the default LRU accesses to change the cache management. Cache management becomes collaborative: the default LRU still considers the past history, and the non-LRU variation enables software control based on future information. The non-LRU access is called a software *hint*.[1]

[1] LRU accesses are also hints since collaborative caching selects both types of accesses, and the optimal management comes out of the combined effect.

There are two questions with this approach: how much the collaboration via non-LRU accesses can improve over traditional cache and how these special accesses may affect the formal properties of cache, whether they are used judiciously or not.

The chapter studies three designs of collaborative cache, each with a different type of non-LRU access.

- *LRU-MRU cache.* The cache is bipartite where the data blocks accessed normally are managed by LRU, and the data blocks accessed by non-LRU are managed by MRU (which replaces the most recently accessed data upon eviction). The MRU access is a bypass in that the data block will not stay in cache (if the cache size is smaller than the data size).
- *Trespass LRU cache.* The non-LRU access is a trespass in that it causes the cache to evict the *most* recently accessed data block rather than the least recently accessed data block as in the default policy.
- *Priority LRU cache.* The accessed data block is associated with a numerical priority for hardware to prioritize its cache management.

The first two types are intended for practical implementation. They change the default LRU design only peripherally. The third type is a generalization that can implement all known inclusive caching algorithms, collaborative or not.

For each type, the chapter examines two formal properties. The first is optimality. We will show that they can all be used to effect optimal cache management. Bypass and trespass hints are specific to a given cache size while the same priority hints can be used to optimize for all cache sizes.

The second is inclusion property. The inclusion property was first characterized by Mattson et al. in their seminal paper in 1970 [41]. The property states that

a larger cache always contains the content of a smaller cache. The property is important for at least three reasons.

i) In inclusive caches, the miss ratio is a monotone function of the cache size. There can be no Belady anomaly [8].

ii) The miss ratio of an execution can be simulated in one pass for all cache sizes, known as the stack simulation (Section 3.2).

iii) Most importantly for software analysis, there exists a distance metric known as stack distances (Section 3.2). An access misses in cache if and only if its stack distance exceeds the cache size. For example, the LRU stack distance has been called the *reuse distance* and used extensively in software and system optimization.

In this chapter, we show that the inclusion property holds for collaborative cache. As an interface, the non-LRU access may be used in arbitrary ways, sometimes optimal but probably suboptimal most times and even counter productive. The inclusion property holds regardless of the usage. The chapter also gives the algorithms to compute the stack distance for collaborative cache: the LRU-MRU distance and the Priority LRU distance.

Being the most general, Priority LRU shows a new type of inclusion which is not uniform. Non-uniform inclusion gives rise to a completely new category of cache not known before this study. It also necessitates a significantly different algorithm to measure its stack distance.

The theoretical findings are the foundation for the application in the next chapter. While in pure theory, we are not concerned with any measure of empirical efficiency, i.e. analysis time and specific cache miss ratio, and will leave these practical considerations for the next chapter.

3.2 Background on Non-collaborative Caching

Mattson's algorithms use a stack to do the simulation and hence are called stack algorithms [41].

Stack simulation provides a metric called stack distance. Stack distance is useful for program analysis because it is independent of specific cache sizes.

An inclusive cache can be viewed as a stack or a priority list. Data elements at the top c stack positions are the ones in a cache of size c.[2] The stack position defines the importance of the stored data. Stack simulation is to simulate cache of an infinite size. *Stack distance* gives the minimal cache size to make an access a cache hit [41]. A stack distance is defined for each type of inclusive cache and computed by simulating that type of cache in an infinite size.

The following are examples of inclusive cache.

Least Recently Used (LRU)

The priority used in LRU cache is the most recent access time. The data element with the least recent access time has the lowest priority (highest position number) and is evicted when a replacement happens. Most hardware implements pseudo-LRU for efficiency [53]. The LRU stack distance is called reuse distance for short. It measures the amount of data accessed between two consecutive uses of the same data element. Reuse distance can be measured in near constant time by organizing the priority list as a dynamically compressed tree [70].

Table 3.1 is an example for LRU cache. The priority list shows the cache content snapshots after each access. For example, the content of a cache of size 3 is the data elements sitting at the top three positions in the priority list. If the data element visited by the next access is in the current top three positions, then

[2] The newly discovered Priority LRU is an exception and will be presented in Section 3.5.

the next access is a hit. Otherwise, it is a miss. For the example trace, there are only three hits with an LRU cache in size 3: access 5, 6, and 7.

The reuse distances are shown at the lowest row in Table 3.1. For example, the reuse distance of No.7 access to data element b is 3 because there are 3 different data elements accessed between the two consecutive accesses to b: No.2 and No.7 accesses. This type of reuse distance is called *backward reuse distance* because the distance looks back from the last access of a reuse pair. On the contrary, *forward reuse distance* looks forward from the first access of a reuse pair. There are only three reuse distances are no greater than 3—only three hits with an LRU cache of size 3.

access time		1	2	3	4	5	6	7	8	9	10	11	12
accessed data		a	b	c	d	d	c	b	a	d	c	b	a
priority	1	a:1	b:2	c:3	d:4	d:5	c:6	b:7	a:8	d:9	c:10	b:11	a:12
list	2		a:1	b:2	c:3	c:3	d:5	c:6	b:7	a:8	d:9	c:10	b:11
	3			a:1	b:2	b:2	b:2	d:5	c:6	b:7	a:8	d:9	c:10
(data-priority)	4				a:1	a:1	a:1	a:1	d:5	c:6	b:7	a:8	d:9
reuse distance		∞	∞	∞	∞	1	2	3	4	4	4	4	4

Table 3.1: An example for LRU cache

The General Rules for Priority List Adjustment

The priority list in the above LRU example is easy to understand because the priority list is ordered. However, LRU is a special case, and a priority list is not necessarily ordered by the used priorities. Two examples of MRU and OPT in Table 3.2 and Table 3.3 shows unordered priority lists.

The general rules for priority list adjustment were first devised by Matterson et al. in their seminal paper [41]. $s_t(i)$ and $v_t(i)$ denote the priority at position i and the victim priority for a cache of size i after No.t access respectively. p_t denotes the priority for No.t access. The general rules for No.t access are:

$$s_t(1) = p_t \tag{3.1}$$

$$v_t(1) = s_{t-1}(1) \tag{3.2}$$

$$s_t(i) = MAX_or_MIN(v_t(i-1), s_{t-1}(i)) \quad (2 \le i \le K) \tag{3.3}$$

$$v_t(i) = MAX_or_MIN(v_t(i-1), s_{t-1}(i)) \quad (2 \le i \le K-1) \tag{3.4}$$

Equations 3.1 and 3.2 make sure that the data element visited by the current access is brought into the cache for all cache sizes. Equations 3.3 and 3.4 make adjustment from No.2 position to No.K position cooperatively. No.K position is the place where the data element visited by the current access sits before the current access happens. All the data elements in the priority list are adjusted if the current access is a compulsory miss—an access to a new data element [29]. MAX_or_MIN is MAX or MIN, which depends on the cache replacement algorithm. The one used in Equation 3.4 is the opposite of the one in Equation 3.3. For LRU, MAX is the one used in Equation 3.3 but MIN in Equation 3.4.

The above equations do not cover the case when $p_t = s_{t-1}(1)$. That case is trivial and no adjustment is needed for the priority list.

Most Recently Used (MRU)

The priorities used in MRU are the same as the ones used in LRU—the most recent access time. Unlike LRU, MIN is the one used in Equation 3.3 and MAX in Equation 3.4.

The priority list for MRU is not necessarily ordered. An example is in Figure 3.2. For the same memory trace, there are five hits with an MRU cache of size 3: access 5, 7, 8, 10, and 11. The stack distance row shows backward MRU

access time		1	2	3	4	5	6	7	8	9	10	11	12
accessed data		a	b	c	d	d	c	b	a	d	c	b	a
priority	1	a:1	b:2	c:3	d:4	d:5	c:6	b:7	a:8	d:9	c:10	b:11	a:12
list	2		a:1	a:1	a:1	a:1	a:1	a:1	b:7	c:6	d:9	d:9	d:9
(data-	3			b:2	b:2	b:2	b:2	c:6	c:6	b:7	b:7	c:10	c:10
priority)	4				c:3	c:3	d:5	d:5	d:5	a:8	a:8	a:8	b:11
MRU stack distance		∞	∞	∞	∞	1	4	3	2	4	3	3	4

Table 3.2: An example for MRU cache

stack distances. There are five distances no greater than 3; five hits with an MRU cache of size 3.

Optimal (OPT)

The priorities used in OPT are the next access time. MAX is the one used in Equation 3.3 and MIN in Equation 3.4. The setup guarantees the in-cache data element with the furthest reuse is the victim if an eviction is needed. OPT is impractical because it requires future knowledge. It serves as the upper bound of cache performance. The fastest method for calculating the OPT stack distance is the one-pass algorithm by Sugumar and Abraham [54].

access time		1	2	3	4	5	6	7	8	9	10	11	12
accessed data		a	b	c	d	d	c	b	a	d	c	b	a
next access time		8	7	6	5	9	10	11	12	∞	∞	∞	∞
priority	1	a:8	b:7	c:6	d:5	d:9	c:10	b:11	a:12	d:∞	c:∞	b:∞	a:∞
list	2		a:8	b:7	c:6	c:6	d:9	d:9	d:9	a:12	a:12	a:12	b:∞
(data-	3			a:8	b:7	b:7	b:7	c:10	c:10	c:10	d:∞	c:∞	c:∞
priority)	4				a:8	a:8	a:8	a:8	b:11	b:11	b:11	d:∞	d:∞
OPT stack distance		∞	∞	∞	∞	1	2	3	4	2	3	4	2

Table 3.3: An example for OPT cache

The priority list for OPT is not necessarily ordered. An example is shown in Figure 3.3. For the same memory trace, there are six hits with an OPT cache of size 3: access 5, 6, 7, 9, 10, and 12. Because OPT is optimal, the maximal

number of hits for the example trace running with a cache of size 3 is six. The stack distance row shows backward OPT stack distances. There are six distances no greater than 3; that is, six hits with an OPT cache of size 3.

3.3 LRU-MRU Cache

3.3.1 Cache Design

In LRU-MRU, an access can be normal LRU or bypass MRU, which are illustrated in Figure 3.1, Figure 3.2, Figure 3.3, and Figure 3.4.

- *Normal LRU access* uses the most recently used position for placement and the least recently used position for replacement

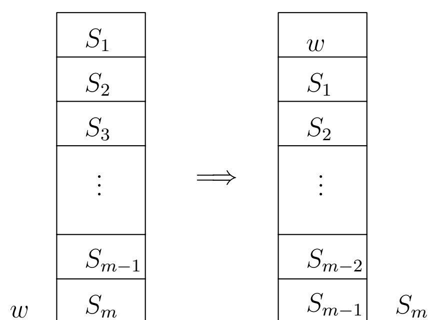


Figure 3.1: Normal LRU at a miss: w is placed at the top of the stack, evicting S_m.

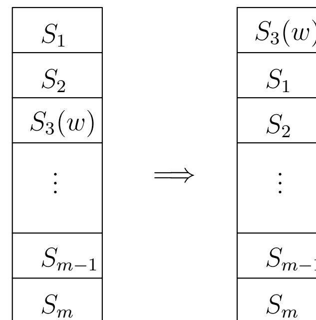


Figure 3.2: Normal LRU at hit: w, assuming at entry S_3, is moved to the top of the stack.

  - Miss: Evict the data element S_m at the LRU position (bottom of the stack) if the cache is full, shift other data elements down by one position, and place w, the visited element, in the MRU position (top of the stack). See Figure 3.1.
  - Hit: Find w in cache, shift the elements over w down by one position, and re-insert w at the MRU position. See Figure 3.2. Note that search

cost is constant in associative cache where hardware checks all entries in parallel.

- *Bypass MRU access* uses the LRU position for placement and the same position for replacement. It is similar to the bypass instruction in IA64 [5] except that its bypass demotes the visited element to LRU position when hit.

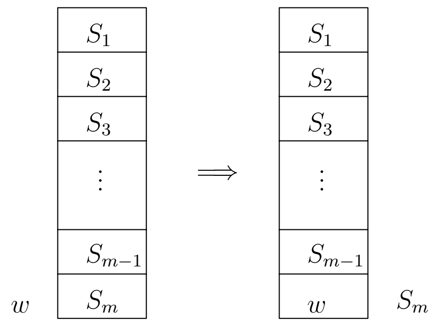


Figure 3.3: Bypass MRU at a miss: the bypass posits w at the bottom of the stack, evicting S_m.

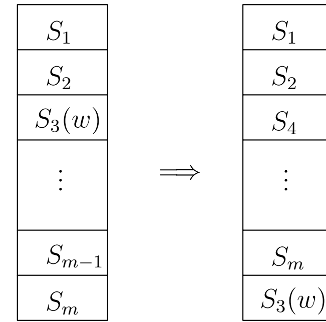


Figure 3.4: Bypass MRU at a hit: the bypass moves $S_3(w)$ to the bottom of the stack.

  - Miss: Evict S_m at the LRU position if the LRU position is taken and insert w into the LRU position. See Figure 3.3.
  - Hit: Find w, lift the elements under w by one position, and place w in the LRU position. See Figure 3.4.

LRU-MRU cache differs from conventional cache in three ways:

- *LRU-MRU content.* The cache stack is divided into two parts: the upper part for LRU data and the lower part for MRU data. Either part may be missing, and the cache is entirely LRU or MRU.

- *Capacity-dependent placement.* The MRU data is placed at the bottom. The location depends on the size of the cache.

- *Hybrid priority.* The LRU part is prioritized by the LRU order; that is, the last accessed is last replaced. The MRU part is by the MRU order; that is, the last accessed is first replaced.

In comparison, conventional non-collaborative cache manages data using a single priority order, for example, LRU by the last access time and OPT by the next access time. The placement depends on the priority order and not on cache size. The single priority naturally gives rise to the inclusion property and its practical benefits.

In LRU-MRU cache, an access could be either a normal LRU or a bypass MRU. A 1-bit cache hint indicates whether an access is LRU or MRU. As an interface, it may be used in arbitrary ways, sometimes optimal as shown in Section 3.3.2 and proved in Section 3.3.3 but probably suboptimal in most times and even counter productive. OPT*, a faster implementation of the original OPT, is presented in Section 3.3.5 for the training pass to find optimal hints.

It is known that both LRU and MRU are stack algorithms holding the inclusion property. In Section 3.3.6, it is proved that the inclusion property holds even in an LRU-MRU cache. To calculate the LRU-MRU stack distance, a hybrid priority scheme is devised in Section 3.3.7.

3.3.2 Optimal LRU-MRU Hints

Given an execution trace and an LRU-MRU collaborative cache of size c, the optimal hint for each access is LRU if the forward OPT distance is equal to or less than c; otherwise, it is MRU. For each access, the forward OPT distance is the OPT distance of the following access to the same data element. The OPT distance is infinite if there is no follow up reuse.

As an example, consider the data access trace in Figure 3.5. To make it interesting, the trace has mixed locality: two blocks `xy` are reused frequently,

having high locality; while the other seven blocks `abcdefg` (highlighted in red) have a streaming pattern and low locality, as shown in the first row. The second and third rows show the OPT and the forward OPT distances. Assuming a cache size $c = 5$, the optimal LRU-MRU hints are given in the last row.

trace	xyaxybxycxydxyexyfxygxyaxybxycxydxye
OPT distance	---23-23-23-23-23-23-234235236237238
forward OPT distance	234235236237238239234235236237238239
optimal hint (c=5)	LLLLLLLLMLLMLLMLLMLLLLLLLLMLLMLLMLLM

Figure 3.5: An example of optimal LRU-MRU hint insertion. For each access, the forward OPT distance is the OPT distance for the following access to the same data element, and the optimal hint for each access is LRU if and only if the forward OPT distance is equal to or less than the cache size.

When the cache size is 5, the optimal management chooses to store high-locality data, that is, `xy`, in cache, as indicated by the small (forward) OPT distances 2 and 3. It stores two of the low-locality streaming blocks, which have OPT distances of 4 and 5 in the remaining space. The rest is not stored in cache.

The hints replicates the behavior of the optimal cache on the LRU-MRU cache. High-locality `xy` are accessed by LRU and so are two of the low-locality blocks to utilize the residual space. The rest are by MRU. Through the hints, the simple LRU-MRU cache imitates the optimal cache and stores as much data in the cache as can benefit from it.

As a comparison, Figure 3.6 gives the stack distances for LRU (that is, the reuse distance), MRU, OPT, and LRU-MRU. From the distances we can compute the number of capacity misses for all cache sizes. An access is a miss under a management policy if the distance exceeds the cache size (miss iff $dis > c$).

The distances explain the inner workings of the cache management. The high distances in LRU show its inability to cache any low-locality data. The high distances in MRU show its problem with caching high-locality data. Both OPT and LRU-MRU treat the two working sets separately and always have low dis-

trace	xyaxybxycxydxyexyfxygxyaxybxycxydxye	#miss (c=5)
LRU distance	---33-33-33-33-33-33-119119119119119	5
MRU distance	---23-34-45-56-67-78-892924345456567	10
OPT distance	---23-23-23-23-23-23-234235236237238	3
LRU-MRU distance	---33-33-32-32-32-32-325334336327328	3

Figure 3.6: Comparing LRU, MRU, OPT, and LRU-MRU. LRU-MRU is optimal for the targeted cache size.

tances for high-locality data and linearly increasing distances for low-locality data. The varying distances are effectively priorities through which these policies select program data to cache.

When the cache size is 5, the miss counts are 5 and 10 for LRU and MRU but just 3 for OPT and LRU-MRU. In fact, for the trace shown above, the hints give the optimal performance for any LRU-MRU cache of size 3 or higher. When the size is 2, the LRU-MRU cache has 3 more misses than OPT. To see this, observe that the collection of LRU-MRU distances differs from the collection of OPT distances only in the number of 2s and 3s. The frequency of the occurrences of other distances is the same. There is one 4 and one 5 in both LRU-MRU and OPT but in a different order. This shows that the optimal cache management is not unique. LRU-MRU is optimal but does not always make identical replacement decisions as OPT.

At the trace level, LRU-MRU must be optimal for the targeted cache size, but the same hints may not be optimal for all cache sizes. At program level, however, we may use program transformation to change the LRU-MRU hints to automatically optimize for all cache sizes, which is discussed in Chapter 4.

3.3.3 Optimality

The type of each access is determined in the OPT training. The details to obtain the optimal cache hints are shown in Figure 3.7. We run an offline OPT simulation

on a trace from a_1 to a_n with a given cache size. At a_j, we find out that the data element X is evicted. Then a_i, the most recent access to X, is selected to use bypass. After training, the unselected accesses still use normal LRU. The training result is specific to the cache size being used.

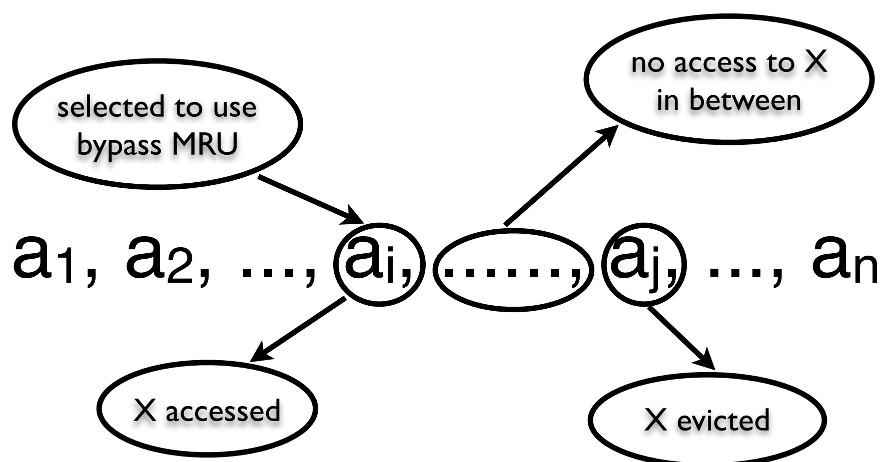


Figure 3.7: a_i is selected for bypass for a given cache size during an OPT cache simulation.

An example of optimal LRU-MRU is shown in Table 3.4 to demonstrate that optimal LRU-MRU has the same result as OPT.

Thursday, June 13, 13

accessed time	1	2	3	4	5	6	7	8	9	10	11
accessed data	a	b	c	d	d	c	e	b	e	c	d
OPT	a	b	c	d	d	c	e	b	e	c	d
cache		a	b	c	c	d	c	e	b	b	b
misses	X	X	X	X			X	X		X	X
bypasses	X	X			X	X			X	X	
optimal			c	d	c	d	**e**	b	b	b	d
LRU-MRU cache	a	b		c	d	c	**d**	e	e	c	b
misses	X	X	X	X			X	X		X	X

Table 3.4: An example showing optimal LRU-MRU with cache size 2

We next prove the optimality for all traces.

Lemma 3.1. *If the bottom element in the optimal LRU-MRU stack is most recently visited by a normal access, then all cache elements are most recently visited by some normal accesses.*

Proof. If some data elements are most recently visited by bypass accesses, then they appear only at the bottom of the stack. They can occupy multiple positions but cannot be lifted up over an element most recently visited by a normal access. Therefore, if the bottom element is most recently visited by a normal access, all elements in the cache must also be. □

Theorem 3.1. *Optimal LRU-MRU generates no more misses than OPT. In particular, optimal LRU-MRU has a miss only if OPT has a miss.*

Proof. We show that there is no access that is a cache hit in OPT but a miss in optimal LRU-MRU. Suppose the contrary were true. Let $z\prime$ be the first access in the trace that hits in OPT but misses in optimal LRU-MRU. Let d be the element accessed by $z\prime$, z be the most recent access to d before $z\prime$, and the reference trace between them be $(z, ..., z\prime)$. The access z can be one of the two cases:

- z is a normal access. For z' to miss in optimal LRU-MRU, there should be a miss y in $(z, ..., z\prime)$ that evicts d. From the assumption that $z\prime$ is the earliest access that is a miss in optimal LRU-MRU but a hit in OPT, y must be a miss in OPT. Consider the two possible cases of y:
  - y occurs when the OPT cache is partially full. Because the OPT cache is always full after the loading of the first M elements, where M is the cache size, this case can happen only at the beginning. However, when the cache is not full, OPT will not evict any element. Hence this case is impossible.
  - y occurs when the OPT cache is full. The element d is at the LRU position before the access of y. According to Lemma 3.1, the optimal

LRU-MRU cache is full, and the most recent accesses of all data elements in cache are normal accesses. Let the set of elements in cache be T for optimal LRU-MRU and T^* for OPT. At this time (before y), the two sets must be identical. The reason is a bit tricky. If there is an element d' in the optimal LRU-MRU cache but not in the OPT cache, d' must be replaced by OPT before y. However, by the construction of the algorithm, the previous access of d' before y should be labeled a bypass access. This contradicts to the lemma, which says the most recent access of d' (and all other elements in T) is normal. Since both caches are full, they must be identical; as a result, we have $T = T^*$. Finally, y, in the case of OPT, must evict some element. However, evicting any element other than d would violate our lemma. Hence, such a y cannot exist and this case is impossible.

- z is a bypass access in optimal LRU-MRU. There must be an access $y \in (z, ..., z\prime)$ in the case of OPT that evicts d; otherwise z cannot be designated as a bypass. However, in this case, the next access to d, $z\prime$ cannot be a cache hit in OPT, contradicting the assumption that $z\prime$ is a cache hit in OPT.

Considering both cases, it is impossible for the same access to be a hit in OPT but a miss in optimal LRU-MRU. □

3.3.4 Multi-size Optimality

If we find LRU-MRU for each cache size, we have the optimal LRU-MRU for all sizes. It has to use different hints for different cache sizes. If we view multi-size optimal LRU-MRU as a single policy, it does not have the inclusion property, which is shown using a counter example. By comparing the two examples of optimal LRU-MRU in Table 3.4 and Table 3.5, we see that at the first access to

e, the stack content, given in bold letters, is (**e,d**) in the smaller cache and (**e, c, b**) in the larger cache. Hence the inclusion property does not hold [41].

accessed time	1	2	3	4	5	6	7	8	9	10	11
accessed data	a	b	c	d	d	c	e	b	e	c	d
OPT	a	b	c	d	d	c	e	b	e	c	d
		a	b	c	c	d	c	e	b	e	e
cache			a	b	b	b	b	c	c	b	b
misses	X	X	X	X			X				X
bypasses	X				X					X	
optimal		b	c	d	c	c	**e**	b	e	e	d
LRU-MRU			b	c	b	b	**c**	e	b	b	e
cache	a			b	d	d	**b**	c	c	c	b
misses	X	X	X	X			X				X

Table 3.5: An example of LRU-MRU with cache size 3. The memory trace is the same one used in Table 3.4. The two examples together show that multi-size optimal LRU-MRU does not have the inclusion property.

Because OPT is optimal, we have the immediate corollary that multi-size optimal LRU-MRU has the same number of misses as OPT and is therefore optimal. In fact, the misses happen for the same accesses in multi-size optimal LRU-MRU and in OPT. Lastly, we show that multi-size optimal LRU-MRU has a peculiar feature.

Corollary 3.1. *Multi-size optimal LRU-MRU does not have the inclusion property, but it does not suffer from Belady anomaly [8], in which the number of misses sometimes increases when the cache size becomes larger.*

Proof. OPT is a stack algorithm since the stack content for a smaller cache is a subset of the stack content for a larger cache [41]. The number of misses of an access trace does not increase with the cache size. Because multi-size optimal LRU-MRU has the same number of misses as OPT, it has the same number of misses as OPT and does not suffer from Belady anomaly. □

3.3.5 The OPT* Algorithm

A faster implementation of OPT called OPT* is designed to do the OPT training. OPT* is asymptotically faster than the original OPT.

Given a memory access sequence, the original OPT algorithm has two passes [41]:

- First pass: Compute the forward reuse distance for each access through a backward scan of the trace.
- Second pass: Incrementally maintain a priority list based on the forward reuse distance of the cache elements. The pass has two steps. First, if the visited element is not in cache, find its place in the sorted list based on its forward reuse distance. Second, after each access, update the forward reuse distance of each cache element.

The update operation is costly and unnecessary. To maintain the priority list, it is sufficient to use the next access time instead of the forward reuse distance. At each point p in the trace, the next access time of data x is the logical time of the next access of x after p. Because the next access time of data x changes only at each access of x, OPT* stores a single next access time at each access in the trace, which is the next access time of the element being accessed. OPT* collects next access times through a single pass traversal of the trace. The revised algorithm OPT* is as follows.

- First pass: Store the next reuse time for each access through a backward scan of the trace.
- Second pass: Maintaining the priority list based on the next reuse time. It has a single step. If the visited element is not in cache, find its place in the sorted list based on its next access time.

The cost per operation is $O(\log M)$ for cache of size M, if the priority list is maintained as a heap. It is asymptotically more efficient than the $O(M)$ per access cost of OPT. The difference is computationally significant when the cache is large. While OPT* is still costly, it is used only for pre-processing and adds no burden to on-line cache management.

OPT* is an enhanced implementation of the original OPT. In the following discussion, we still use OPT to name the training pass since any OPT implementation works.

3.3.6 Inclusion Property

If the collaborative cache is used optimally, the performance is the same as OPT shown in Section 3.3.3. In general, however, the cache may not be used optimally. The selection of MRU accesses may be arbitrary. The following proof is for all uses of LRU-MRU cache, including the extreme cases (when all accesses are normal, i.e. LRU caching, and when all accesses are special, i.e. MRU caching), the optimal use, and everything in between. In a sense, the proof subsumes the individual conclusions for LRU, MRU, and OPT [41].

We prove that for any sequence of LRU and MRU accesses, the LRU-MRU cache obeys the inclusion principle.

Lemma 3.2. *If the bottom element in the LRU-MRU cache stack is most recently accessed by a normal LRU access, then all elements in cache are most recently accessed by normal LRU accesses.*

The Lemma 3.2 follows from the fact that MRU data are placed at the bottom of the stack and only replaced by LRU data (never pushed up except by other MRU data). The formal proof is in Lemma 3.1 on page 31. Next we prove the inclusion property.

Theorem 3.2. *A trace P is being executed on two LRU-MRU caches of sizes $|C_1|$ and $|C_2|$ ($|C_1| < |C_2|$). At every access, the content of cache C_1 is always a subset of the content of cache C_2.*

Proof. Let the access trace be $P = (x_1, x_2, ..., x_n)$. Let $C_1(t)$ and $C_2(t)$ be the set of elements in cache C_1 and C_2 after access x_t. The data element visited by x_t is $d(x_t)$. The initial cache contents are $C_1(0) = C_2(0) = \emptyset$. The inclusion property holds. We now prove the theorem by induction on t.

Assume $C_1(t) \subseteq C_2(t)$ $(0 \leq t \leq n-1)$. It is easy to see that if x_{t+1} is a hit in C_2 ($x_{t+1} \in C_2(t)$), the inclusion property holds. We now consider the case that x_{t+1} is a miss in C_2. Since C_1 is included in C_2, x_{t+1} is also a miss in C_1.

Let the evicted elements be most recently accessed at x_p in C_1 and x_q in C_2. After the cache miss, we have $C_1(t+1) = C_1(t) - d(x_p) + d(x_{t+1})$ and $C_2(t+1) = C_2(t) - d(x_q) + d(x_{t+1})$. Since $C_1(t) \subseteq C_2(t)$, the only possibility for $C_1(t+1) \nsubseteq C_2(t+1)$ is that C_2 evicts $d(x_q)$, and C_1 has $d(x_q)$ but does not evict it, so $d(x_q) \in C_1(t+1)$ but $d(x_q) \notin C_2(t+1)$.

First, we assume $d(x_p)$ exists (a cache miss does not mean a cache eviction. see the next case). The eviction in C_1 happens at the LRU position regardless of whether x_p is a LRU or MRU access. So, $d(x_p)$ is at the bottom in C_1 before access x_{t+1}. At the same time, $d(x_q)$ is at the bottom in C_2. To violate the inclusion property, we must have $d(x_q) \in C_1(t)$ in a position over $d(x_p)$. From the inductive assumption, $d(x_p) \in C_2(t)$, and it is in a position over $d(x_q)$. Therefore, both C_1 and C_2 contain $d(x_p)$ and $d(x_q)$ but in an opposite order.

The two accesses, x_p and x_q, may be LRU or MRU accesses. There are four cases:

I. x_p and x_q are both LRU accesses. Because $d(x_q)$ is at a higher position than $d(x_p)$ in C_1, we have $p < q$. Similar reasoning from C_2 requires $q < p$, which makes this case impossible.

II. x_p is an LRU access but x_q is an MRU access. Using Lemma 3.2 on C_1, we see that this case is impossible—x_q has to be an LRU access because $d(x_q)$ resides over $d(x_p)$ that is most recently accessed by an LRU access in C_1.

III. x_p is an MRU access but x_q is an LRU access. Using Lemma 3.2 on C_2, we see that this case is impossible—x_p has to be an LRU access because $d(x_p)$ resides over $d(x_q)$ that is most recently accessed by an LRU access in C_2.

IV. x_p and x_q are both MRU accesses. Because $d(x_q)$ is at a higher position than $d(x_p)$ in C_1, we have $p > q$. Similar reasoning from C_2 requires $q > p$, which makes the last case impossible.

There is no eviction in C_1 if the bottom cache line is unoccupied when x_{t+1} is accessed, and $d(x_q)$ is at the bottom of C_2. Regardless of whether x_q is LRU or MRU, C_2 is filled. Because $|C_2| > |C_1|$, there must have been enough data access to fill C_1, making it impossible for its bottom spot to remain unoccupied. Hence, by induction, the inclusion property holds for every access in the trace. □

The inclusion property holds for any access trace with mixed LRU and MRU accesses, regardless of how these two types of accesses are interleaved.

3.3.7 LRU-MRU Stack Distance

The inclusion property implies the existence of the LRU-MRU stack distance. An access has a distance k if it is a cache hit in caches of sizes k and up and a miss in caches of size $k - 1$ and down. Given a program trace with mixed LRU-MRU accesses, Algorithm 3.1 computes the stack distance for each access. Effectively, the algorithm simulates LRU-MRU caches of *all* sizes—top C elements in the priority list are always the content of a cache with size C. We call the algorithm *bi-sim* in short for LRU-MRU cache simulation.

Algorithm 3.1: Bi-sim: computing the stack distance of LRU-MRU cache

Input: x is accessed at time t with flag $f = \{LRU, MRU\}$. The cache is organized as a priority list, with data d_i and priority p_i, $i = 1, \ldots, m$. No two priorities are the same, that is, $\forall i$ and j, $p_i \neq p_j$ if $i \neq j$. The list may not have been sorted.

Output: It returns the LRU-MRU stack distance and updates the priority p_x of x (first adding it to the priority list if it was not included). The priority p_x is unique.

```
1  bi_sim(x,t,f)
2  begin
3      if f == LRU then
4          p_x = t
5      else
6          p_x = −t
7      end
8      /* adjust the priority list */
9      if x ∉ {d_i : i = 1, . . . , m} then
10         /* x is a miss */
11         for i = 1; i < M; i++ do
12             if p_i < p_{i+1} then
13                 Swap d_i and d_{i+1}
14             end
15         end
16         /* d_m is at the bottom of the cache */
17         if p_m < 0 then
18             Remove d_m from the list
19         end
20         Insert x at the front of the list
21         Return ∞
22     else
23         /* x is a hit */
24         Find out d_k = x
25         for i = 1; i < k; i++ do
26             if p_i < p_{i+1} then
27                 Swap d_i and d_{i+1}
28             end
29         end
30         Move x to the front of the list
31         Return k
32     end
33 end
```

For access x at time t, Algorithm 3.1 computes the stack distance and updates the priority list. The algorithm has three parts:

- The first part, lines 3 to 7, sets the priority for x to be t or $-t$ depending on whether x is LRU or MRU. The purpose is to handle mixed priority. By negating t, the priority of MRU data is reversed to the access order. The MRU in the access order becomes LRU in the priority order. In addition, the negative priority means that all MRU data has a lower priority than all LRU data. Finally, all priority numbers remain distinct. As a result, all data in the cache are prioritized with no ties.

- The second part, lines 11 to 21, handles cache replacement at a miss when x is not in the priority list. The element with the lowest priority is shifted down to the bottom. It is removed if its priority is negative (an MRU data element). Element x is inserted to become the new head of the list.

- The third part, lines 24 to 31, handles a hit at location k, that is, $d_k = x$. The element of the lowest priority in $d_1, \ldots, d_k$ is shifted down to replace d_k. Element x is moved to the front of the list as in the second part.

The update process, swapping and then inserting, is similar to Mattson et el. [41] but with two notable qualities. First, the priority list of bi-sim is not completely sorted. In comparison, the priority list in LRU simulation is always totally sorted. Second, bi-sim may remove an element from the priority list (line 16), even if it is simulating a cache of an infinite size. The stack simulation of previous caching methods such as LRU and OPT never removes elements when simulating for all cache sizes.

An example An example depicting bi-sim in action is given in Table 3.6. The access trace and the access types are listed in the second and third columns. The

priority list (after each access) is shown in the next column. The last column is the stack distance returned by Algorithm 3.1: ∞ always means a miss, and k means a cache hit if cache size $C \geq k$ and a miss otherwise. The priority lists in the table show only the priority numbers p_x. A reader can find the data element from the p_xth row of the table (the p_xth access in the trace).

The example shows two notable characteristics of the bi-sim algorithm. The priority list is not completely sorted because of the negative priority numbers of MRU accesses. An MRU element may be removed from cache even when there is space, as happens at access 3. These are necessary to measure the miss ratios of all cache sizes in a single pass.

The cost and its reduction The asymptotic cost of Algorithm 3.1 is $O(M)$ in time and space for each access, where M is the number of distinct data elements in the input trace. The main time overhead comes from the two swap loops at lines 11-15 and 25-29. To improve performance, we divide the priority list into partially sorted groups. For example, there are 4 groups at the 25th access in the example in Table 3.6: [26], [22, -23], [20, -24], and [17, 14, 11, 1][3]. The swap loops are changed to iterate over the groups. The minimal element of a group is simply the last element. Grouping in priority lists was first invented by Sugumar and Abraham for simulating OPT [54]. A difference between OPT and bi-sim is that the accessed data element can be in the middle of a group in bi-sim. For OPT, the accessed data element always stays at the front of a group.

The Equivalence Proof

So far we have presented the LRU-MRU cache and its simulation. We now show that the simulation algorithm is correct; that is, the elements of the priority list $d_1, d_2, \ldots, d_C$ in the algorithm are indeed the content of an LRU-MRU cache of

[3]For convenience, the top element is always put into a separate window.

access no.	access trace	LRU or MRU	the priority list (top → bottom)									stack distance
1	h	L	1									∞
2	f	M	-2	1								∞
3	i	L	3	1								∞
4	i	M	-4	1								1
5	c	L	5	1								∞
6	b	L	6	5	1							∞
7	b	M	-7	5	1							1
8	e	M	-8	5	1							∞
9	d	M	-9	5	1							∞
10	b	L	10	5	1							∞
11	g	L	11	10	5	1						∞
12	b	L	12	11	5	1						2
13	e	L	13	12	11	5	1					∞
14	d	L	14	13	12	11	5	1				∞
15	a	L	15	14	13	12	11	5	1			∞
16	c	L	16	15	14	13	12	11	1			6
17	e	L	17	16	15	14	12	11	1			4
18	a	L	18	17	16	14	12	11	1			3
19	c	L	19	18	17	14	12	11	1			3
20	i	L	20	19	18	17	14	12	11	1		∞
21	f	L	21	20	19	18	17	14	12	11	1	∞
22	b	L	22	21	20	19	18	17	14	11	1	7
23	a	M	-23	22	21	20	19	17	14	11	1	5
24	f	M	-24	22	-23	20	19	17	14	11	1	3
25	c	M	-25	22	-23	20	-24	17	14	11	1	5
26	c	L	26	22	-23	20	-24	17	14	11	1	1
27	e	M	-27	26	22	20	-23	-24	14	11	1	6
28	i	M	-28	26	22	-27	-23	-24	14	11	1	4
29	c	L	29	-28	22	-27	-23	-24	14	11	1	2
30	f	L	30	29	22	-27	-23	-28	14	11	1	6

Table 3.6: Example one-pass simulation of LRU-MRU cache

size C. We show the equivalence in two steps. First, we show that the algorithm observes the inclusion property. Then we show that the two are equivalent at each cache size.

Proving the inclusion property is easier for the algorithm than for LRU-MRU cache because we can use its algorithmic design directly. We first define a property in cache replacement. Let two caches of size $s, s+1$ be C_s, C_{s+1}, which are also the data sets in cache. Assume that C_s, C_{s+1} are filled with data, and z is the element in C_{s+1} but not in C_s. At a cache miss, C_s evicts element y_s, and C_{s+1} evicts y_{s+1}. The *eviction invariance* is a property that requires

$$y_{s+1} = y_s \vee y_{s+1} = z$$

Mattson et al. [41] showed the following result:

Lemma 3.3. *Eviction invariance is a necessary and sufficient condition for maintaining the inclusion property.*

Proof. First, we show the necessity. If $y_{s+1} \neq y_s \wedge y_{s+1} \neq z$, y_{s+1} must be in C_s. Its eviction would mean that $C_s \nsubseteq C_{s+1}$ and would break the inclusion property. The property is also sufficient. At each eviction, if $y_{s+1} = y_s$, we have $C_{s+1} = C_s + z$; otherwise, we have $y_{s+1} = z$ and $C_{s+1} = C_s + y_s$. In both cases, $C_s \subseteq C_{s+1}$. □

The simulation algorithm observes the eviction invariance. The "stack" is embodied in a priority list. Each element has a numerical priority distinct from others. Therefore, the caches it simulates have the inclusion property.

Lemma 3.4. *Algorithm 3.1 observes the eviction invariance and is therefore a stack algorithm.*

Proof. Algorithm 3.1 identifies a victim for replacement using one of the two swap loops at lines 11-15 and 25-29. Consider two caches C_s, C_{s+1} of sizes $s, s+1$. Let

z be the element in C_{s+1} but not in C_s. Let y be the element in C_s that has the lowest priority. When a cache replacement is needed in C_{s+1}, the swap loops would choose as the victim y if $p_y < p_z$ and z otherwise. The eviction invariance is therefore observed. □

Intuitively, the simulation is a stack algorithm because the simulated caches of all sizes share a single priority list. It is obvious that sharing a priority list implies eviction invariance. Next we show that Algorithm 3.1 computes the right stack distance. First we have the following lemma. We omit the proof, which is straightforward based on the handling of LRU and MRU accesses.

Lemma 3.5. *At a miss in LRU-MRU cache, the victim is always the data element with the lowest priority.*

Theorem 3.3. *Given an execution on LRU-MRU cache of size C, an access is a cache hit if and only if the stack distance returned by Algorithm 3.1 is no greater than C.*

Proof. The case for infinite distances is easy to verify; we only prove the case when the distance is of a finite value. Specifically, Algorithm 3.1 always stores the data in the priority list such that a cache of size C would contain and only contain the first C elements in the list, $d_1, d_2, \cdots, d_C$. This is equivalent to showing that for each data d_i, we have $d_i \in C_i$ and $d_i \notin C_{i-1}$, where $i > 0$ and C_i, C_{i-1} are the sets of data in caches of sizes $i, i-1$, respectively.

Let the memory trace be $(x_1, x_2, \cdots, x_n)$. We prove by induction on x_j.

I. After accessing x_1, x_1 becomes d_1 in the priority list. The base case holds since $d_1 \in C_1$ and $d_1 \notin C_0$.

II. Assume the theorem holds after accessing x_j $(1 \leq j \leq n-1)$. Let the data element at position i be $d_i(j)$ and the data sets of caches of size $i-1$ and i

be $C_{i-1}(j)$ and $C_i(j)$. From the inductive hypothesis, we have $d_i(j) \in C_i(j)$ and $d_i(j) \notin C_{i-1}(j)$. There are two cases after accessing x_{j+1}:

(a) x_{j+1} is a (compulsory) miss. Each data element of the priority list is updated from $d_i(j)$ to $d_i(j+1)$ ($1 \le i \le m$ or $1 \le i \le m+1$).

   i. $d_1(j+1) = x_{j+1}$ and satisfies $d_1(j+1) \in C_1(j+1)$ and $d_1(j+1) \notin C_0(j+1)$.

   ii. For $d_i(j+1)$ ($2 \le i \le m$), the swap loop (lines 11-15) moves the data element $d_h(j)$ ($1 \le h \le i$) with the lowest priority in $C_i(j)$ out of the priority list. According to Lemma 3.5, after evicting $d_h(j)$ from $C_i(j)$, the top i elements in the priority list are still in $C_i(j+1)$, so $d_i(j+1) \in C_i(j+1)$. In the same way, we can show that $d_i(j+1)$ is either $d_i(j)$ or the victim of $C_{i-1}(j)$, so $d_i(j+1) \notin C_{i-1}(j+1)$.

   iii. If $d_m(j)$ has a positive priority, $d_{m+1}(j+1)$ is at the new bottom and must be the victim of $C_i(j)$, so $d_{m+1}(j+1) \notin C_m(j+1)$. $d_{m+1}(j+1) \in C_{m+1}(j+1)$ follows from Lemma 3.2.

   iv. If $d_m(j)$ has a negative priority, the stack distance would be infinite. It is a miss in all finite-size LRU-MRU cache.

(b) x_{j+1} is a hit. Let the hit location be k ($d_k(j) = x_{j+1}$). Each data element of the priority list is updated from $d_i(j)$ to $d_i(j+1)$ ($1 \le i \le m$).

   i. Consider $d_i(j+1)$ ($1 \le i \le k-1$). The access is a miss in caches $C_1(j), \cdots, C_{k-1}(j)$, so the inference of the previous miss case can be reused here. The swap loop in lines 25-29 is identical to the swap loop in lines 11-15.

   ii. Consider $d_k(j+1)$. Because $C_k(j) = C_k(j+1)$, we have $d_k(j+1) \in C_k(j+1)$. From the inference of the miss case, $d_k(j+1) \notin C_{k-1}(j+1)$.

iii. Finally, consider $d_i(j+1)$ $(k+1 \leq i \leq m)$, $d_i(j+1) = d_i(j)$ because there is no change made by the algorithm. From $x_{j+1} = d_k(j) \in C_k(j)$, we have x_{j+1} is a cache hit in $C_i(j)$ $(i \geq k+1)$ and $C_i(j) = C_i(j+1)$ $(k+1 \leq i \leq m)$. From the induction assumption, we have $d_i(j+1) \in C_i(j+1)$ and $d_i(j+1) \notin C_{i-1}(j+1)$ $(k+1 \leq i \leq m)$.

For all accesses, the cache of size C would contain and only contain the first C elements in the priority list, $d_1, d_2, \cdots, d_C$. Hence the relationship is established between the stack distance and the cache hit/miss as stated in the theorem. □

3.4 Trespass LRU Cache

Trespass LRU is very similar to optimal LRU-MRU. In Trespass LRU, an access can be normal LRU or trespass MRU, which are illustrated in Figure 3.1, Figure 3.2, Figure 3.8, and Figure 3.9.

- *Trespass MRU access* uses the most recently used position for placement and the same position for replacement. It differs from all cache replacement policies that we are aware of in that an eviction may happen even for a cache hit.
  - Miss: Evict the data element S_1 at the MRU position if the MRU is taken and insert w in the MRU position. See Figure 3.8.
  - Hit: If w is in the MRU position, then do nothing. Otherwise, evict the data element S_1 at the MRU position, insert w there, and shift the elements under the old w spot up by one position. See Figure 3.9.

Trespass LRU is proved optimal in Section 3.4.1. Section 3.4.2 shows that multi-size optimal Trespass LRU holds the inclusion property.

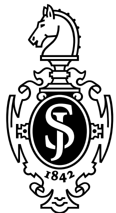
Lecture Notes in Computer Science 2625
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Ulrich Meyer   Peter Sanders   Jop Sibeyn (Eds.)

Algorithms for Memory Hierarchies

Advanced Lectures



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Ulrich Meyer
Peter Sanders
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
E-mail: {umeyer,sanders}@mpi-sb.mpg.de

Jop Sibeyn
Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik
Von-Seckendorff-Platz 1, 06120 Halle, Germany
E-mail:jopsi@informatik.uni-halle.de

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek.
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): F.2, E.5, E.1, E.2, D.2, D.4, C.2, G.2, H.2, I.2, I.3.5

ISSN 0302-9743
ISBN 3-540-00883-7 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2003
Printed in Germany

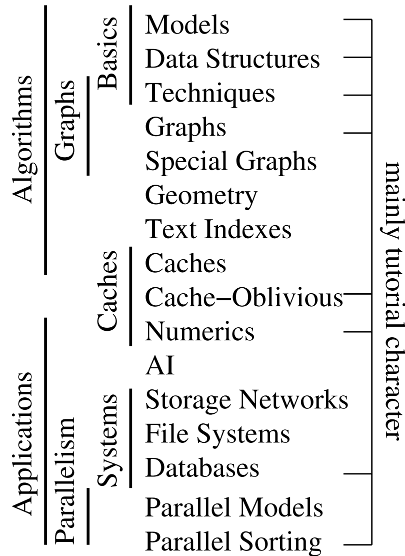
Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10873015 06/3142 5 4 3 2 1 0

Preface

Algorithms that process large data sets have to take into account that the cost of memory accesses depends on where the accessed data is stored. Traditional algorithm design is based on the von Neumann model which assumes uniform memory access costs. Actual machines increasingly deviate from this model. While waiting for a memory access, modern microprocessors can execute 1000 additions of registers. For hard disk accesses this factor can reach seven orders of magnitude. The 16 chapters of this volume introduce and survey algorithmic techniques used to achieve high performance on memory hierarchies. The focus is on methods that are interesting both from a practical and from a theoretical point of view.

This volume is the result of a *GI-Dagstuhl Research Seminar*. The Gesellschaft für Informatik (GI) has organized such seminars since 1997. They can be described as "self-taught" summer schools where graduate students in cooperation with a few more experienced researchers have an opportunity to acquire knowledge about a current topic of computer science. The seminar was organized as Dagstuhl Seminar 02112 from March 10, 2002 to March 14, 2002 in the International Conference and Research Center for Computer Science at Schloss Dagstuhl.

Chapter 1 gives a more detailed motivation for the importance of algorithm design for memory hierarchies and introduces the models used in this volume. Interestingly, the simplest model variant — two levels of memory with a single processor — is sufficient for most algorithms in this book. Chapters 1–7 represent much of the algorithmic core of external memory algorithms and almost exclusively rely on this simple model. Among these, Chaps. 1–3 lay the foundations by describing techniques used in more specific applications. Rasmus Pagh discusses data structures like search trees, hash tables, and priority queues in Chap. 2. Anil Maheshwari and Norbert Zeh explain generic algorithmic approaches in Chap. 3. Many of these techniques such as time-forward processing, Euler tours, or list ranking can be formulated in terms of graph theoretic concepts. Together with Chaps. 4 and 5 this offers a comprehensive review of external graph algorithms. Irit Katriel and Ulrich Meyer discuss fundamental algorithms for graph traversal, shortest paths, and spanning trees that work for many types of graphs. Since even simple graph problems can be difficult to solve in external memory, it



makes sense to look for better algorithms for frequently occurring special types of graphs. Laura Toma and Norbert Zeh present a number of astonishing techniques that work well for planar graphs and graphs with bounded tree width.

In Chap. 6 Christian Breimann and Jan Vahrenhold give a comprehensive overview of algorithms and data structures handling geometric objects like points and lines — an area that is at least as rich as graph algorithms. A third area of again quite different algorithmic techniques are string problems discussed by Juha Kärkäinen and Srinivasa Rao in Chap. 7.

Chapters 8–10 then turn to more detailed models with particular emphasis on the complications introduced by hardware caches. Beyond this common motivation, these chapters are quite diverse. Naila Rahman uses sorting as an example for these issues in Chap. 8 and puts particular emphasis on the often neglected issue of TLB misses. Piyush Kumar introduces *cache-oblivious algorithms* in Chap. 9 that promise to grasp multilevel hierarchies within a very simple model. Markus Kowarschik and Christian Weiß give a practical introduction into cache-efficient programs using numerical algorithms as an example. Numerical applications are particularly important because they allow significant instruction-level parallelism so that slow memory accesses can dramatically slow down processing.

Stefan Edelkamp introduces an application area of very different character in Chap. 11. In artificial intelligence, search programs have to handle huge state spaces that require sophisticated techniques for representing and traversing them.

Chapters 12–14 give a system-oriented view of advanced memory hierarchies. On the lowest level we have storage networks connecting a large number of inhomogeneous disks. Kay Salzwedel discusses this area with particular

emphasis on the aspect of inhomogeneity. File systems give a more abstract view of these devices on the operating system level. Florin Isaila explains the organization of modern file systems in Chap. 13. An even higher level view is offered by relational database systems. Josep Larriba-Pey explains their organization in Chap. 14. Both in file systems and databases, basic algorithmic techniques like sorting and search trees turn out to be relevant.

Finally, Chaps. 15 and 16 give a glimpse on memory hierarchies with multiple processors. Massimo Coppola and Martin Schmollinger introduce abstract and concrete programming models like BSP and MPI in Chap. 15. Dani Jimenez, Josep-L. Larriba, and Juan J. Navarro present a concrete case study of sorting algorithms on shared memory machines in Chap. 16. He studies programming techniques that avoid pitfalls like true and false sharing of cache contents.

Most chapters in this volume have partly tutorial character and are partly more dense overviews. At a minimum Chaps. 1, 2, 3, 4, 9, 10, 14, and 16 are tutorial chapters suitable for beginning graduate-level students. They are sufficiently self-contained to be used for the core of a course on external memory algorithms. Augmented with the other chapters and additional papers it should be possible to shape various advanced courses. Chapters 1–3 lay the basis for the remaining chapters that are largely independent.

We are indebted to many people and institutions. We name a few in alphabetical order. Ulrik Brandes helped with sources from a tutorial volume on graph drawing that was our model in several aspects. The International Conference and Research Center for Computer Science in Dagstuhl provided its affordable conference facilities and its unique atmosphere. Springer-Verlag, and in particular Alfred Hofmann, made it possible to smoothly publish the volume in the LNCS series. Kurt Mehlhorn's group at MPI Informatik provided funding for several (also external) participants. Dorothea Wagner came up with the idea for the seminar and advised us in many ways. This volume was also partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

January 2003

Ulrich Meyer
Peter Sanders
Jop Sibeyn

Table of Contents

1. Memory Hierarchies — Models and Lower Bounds
Peter Sanders . . . 1

1.1 Why Memory Hierarchies . . . 1
1.2 Current Technology . . . 2
1.3 Modeling . . . 5
1.4 Issues in External Memory Algorithm Design . . . 9
1.5 Lower Bounds . . . 10

2. Basic External Memory Data Structures
Rasmus Pagh . . . 14

2.1 Elementary Data Structures . . . 15
2.2 Dictionaries . . . 17
2.3 B-trees . . . 19
2.4 Hashing Based Dictionaries . . . 27
2.5 Dynamization Techniques . . . 33
2.6 Summary . . . 35

3. A Survey of Techniques for Designing I/O-Efficient Algorithms
Anil Maheshwari and Norbert Zeh . . . 36

3.1 Introduction . . . 36
3.2 Basic Techniques . . . 37
3.3 Simulation of Parallel Algorithms in External Memory . . . 44
3.4 Time-Forward Processing . . . 46
3.5 Greedy Graph Algorithms . . . 48
3.6 List Ranking and the Euler Tour Technique . . . 50
3.7 Graph Blocking . . . 54
3.8 Remarks . . . 61

4. Elementary Graph Algorithms in External Memory
Irit Katriel and Ulrich Meyer . . . 62

4.1 Introduction . . . 62
4.2 Graph-Traversal Problems: BFS, DFS, SSSP . . . 63

4.3 Undirected Breadth-First Search 66
4.4 I/O-Efficient Tournament Trees 70
4.5 Undirected SSSP with Tournament Trees 73
4.6 Graph-Traversal in Directed Graphs 74
4.7 Conclusions and Open Problems for Graph Traversal 76
4.8 Graph Connectivity: Undirected CC, BCC, and MSF 77
4.9 Connected Components 78
4.10 Minimum Spanning Forest 80
4.11 Randomized *CC* and *MSF* 81
4.12 Biconnected Components 83
4.13 Conclusion for Graph Connectivity 84

5. I/O-Efficient Algorithms for Sparse Graphs
Laura Toma and Norbert Zeh 85

5.1 Introduction 85
5.2 Definitions and Graph Classes 87
5.3 Techniques 89
5.4 Connectivity Problems 91
5.5 Breadth-First Search and Single Source Shortest Paths 93
5.6 Depth-First Search 98
5.7 Graph Partitions 101
5.8 Gathering Structural Information 105
5.9 Conclusions and Open Problems 109

6. External Memory Computational Geometry Revisited
Christian Breimann and Jan Vahrenhold 110

6.1 Introduction 110
6.2 General Methods for Solving Geometric Problems 112
6.3 Problems Involving Sets of Points 119
6.4 Problems Involving Sets of Line Segments 131
6.5 Problems Involving Set of Polygonal Objects 144
6.6 Conclusions 148

7. Full-Text Indexes in External Memory
Juha Kärkkäinen and S. Srinivasa Rao 149

7.1 Introduction 149
7.2 Preliminaries 150
7.3 Basic Techniques 151
7.4 I/O-efficient Queries 155
7.5 External Construction 161
7.6 Concluding Remarks 170

8. Algorithms for Hardware Caches and TLB
Naila Rahman ... 171

8.1 Introduction ... 171
8.2 Caches and TLB ... 173
8.3 Memory Models ... 176
8.4 Algorithms for Internal Memory ... 181
8.5 Cache Misses and Power Consumption ... 185
8.6 Exploiting Other Memory Models: Advantages and Limitations ... 186
8.7 Sorting Integers in Internal Memory ... 189

9. Cache Oblivious Algorithms
Piyush Kumar ... 193

9.1 Introduction ... 193
9.2 The Model ... 195
9.3 Algorithm Design Tools ... 197
9.4 Matrix Transposition ... 199
9.5 Matrix Multiplication ... 201
9.6 Searching Using Van Emde Boas Layout ... 203
9.7 Sorting ... 205
9.8 Is the Model an Oversimplification? ... 209
9.9 Other Results ... 211
9.10 Open Problems ... 211
9.11 Acknowledgements ... 212

10. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms
Markus Kowarschik and Christian Weiß ... 213

10.1 Introduction ... 213
10.2 Architecture and Performance Evaluation of Caches ... 214
10.3 Basic Techniques for Improving Cache Efficiency ... 217
10.4 Cache-Aware Algorithms of Numerical Linear Algebra ... 225
10.5 Conclusions ... 232

11. Memory Limitations in Artificial Intelligence
Stefan Edelkamp ... 233

11.1 Introduction ... 233
11.2 Hierarchical Memory ... 234
11.3 Single-Agent Search ... 235
11.4 Action Planning ... 241
11.5 Game Playing ... 246
11.6 Other AI Areas ... 248
11.7 Conclusions ... 249

12. Algorithmic Approaches for Storage Networks
Kay A. Salzwedel . . . 251

12.1 Introduction . . . 251
12.2 Model . . . 254
12.3 Space and Access Balance . . . 256
12.4 Availability . . . 258
12.5 Heterogeneity . . . 260
12.6 Adaptivity . . . 269
12.7 Conclusions . . . 272

13. An Overview of File System Architectures
Florin Isaila . . . 273

13.1 Introduction . . . 273
13.2 File Access Patterns . . . 274
13.3 File System Duties . . . 276
13.4 Distributed File Systems . . . 280
13.5 Summary . . . 289

14. Exploitation of the Memory Hierarchy in Relational DBMSs
Josep-L. Larriba-Pey . . . 290

14.1 Introduction . . . 290
14.2 What to Expect and What Is Assumed . . . 291
14.3 DBMS Engine Structure . . . 293
14.4 Evidences of Locality in Database Workloads . . . 297
14.5 Basic Techniques for *Locality Exploitation* . . . 298
14.6 Exploitation of Locality by the Executor . . . 300
14.7 Access Methods . . . 307
14.8 Exploitation of Locality by the Buffer Pool Manager . . . 311
14.9 Hardware Related Issues . . . 317
14.10 Compilation for Locality Exploitation . . . 318
14.11 Summary . . . 318

15. Hierarchical Models and Software Tools for Parallel Programming
Massimo Coppola and Martin Schmollinger . . . 320

15.1 Introduction . . . 320
15.2 Architectural Background . . . 321
15.3 Parallel Computational Models . . . 327
15.4 Parallel Bridging Models . . . 328
15.5 Software Tools . . . 338
15.6 Conclusions . . . 352

16. Case Study: Memory Conscious Parallel Sorting
Dani Jiménez-González, Josep-L. Larriba-Pey, and
Juan J. Navarro 355

16.1 Introduction 355
16.2 Architectural Aspects 358
16.3 Sequential and Straight Forward Radix Sort Algorithms 361
16.4 Memory Conscious Algorithms 371
16.5 Conclusions 376
16.6 Acknowledgments 377

Bibliography 379

Index 421

List of Contributors

Editors

Ulrich Meyer
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
uli@uli-meyer.de

Peter Sanders
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
sanders@mpi-sb.mpg.de

Jop F. Sibeyn
Martin-Luther Universität
Halle-Wittenberg
Institut für Informatik
Von-Seckendorff-Platz 1
06120 Halle, Germany
jopsi@informatik.uni-halle.de

Authors

Christian Breimann
Westfälische Wilhelms-Universität
Institut für Informatik
Einsteinstr. 62
48149 Münster, Germany
chr@math.uni-muenster.de

Massimo Coppola
University of Pisa
Department of Computer Science
Via F. Buonarroti 2
56127 Pisa, Italy
coppola@di.unipi.it

Stefan Edelkamp
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee, Gebäude 51
79110 Freiburg, Germany
edelkamp@informatik.uni-freiburg.de

Florin Isaila
University of Karlsruhe
Department of Computer Science
PO-Box 6980
76128 Karlsruhe, Germany
florin@ipd.uni-karlsruhe.de

Dani Jiménez-González
Universitat Politècnica de Catalunya
Computer Architecture Department
Jordi Girona 1-3, Campus Nord-UPC
E-08034 Barcelona, Spain
djimenez@ac.upc.es

Irit Katriel
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
irit@mpi-sb.mpg.de

Juha Kärkkäinen
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
juha@mpi-sb.mpg.de

Markus Kowarschik
Friedrich–Alexander–Universität
Erlangen–Nürnberg
Lehrstuhl für Informatik 10
Cauerstraße 6,
91058 Erlangen, Germany
Markus.Kowarschik@cs.fau.de

Piyush Kumar
State University of New York
at Stony Brook
Department of Computer Science
Stony Brook, NY 11790, USA
piyush@acm.org

Josep-L. Larriba-Pey
Universitat Politècnica de Catalunya
Computer Architecture Department
Jordi Girona 1-3, Campus Nord-UPC
E-08034 Barcelona, Spain
larri@ac.upc.es

Anil Maheshwari
Carleton University
School of Computer Science
1125 Colonel By Drive
Ottawa, Ontario, K1S 5B6, Canada
maheshwa@scs.carleton.ca

Ulrich Meyer
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
uli@uli-meyer.de

Juan J. Navarro
Universitat Politècnica de Catalunya
Computer Architecture Department
Jordi Girona 1-3, Campus Nord-UPC
E-08034 Barcelona, Spain
juanjo@ac.upc.es

Rasmus Pagh
The IT University of Copenhagen
Glentevej 67
2400 København NV, Denmark
pagh@it-c.dk

Naila Rahman
University of Leicester
Department of Mathematics
and Computer Science
University Road
Leicester, LE1 7RH, U. K.
naila@mcs.le.ac.uk

S. Srinivasa Rao
University of Waterloo
School of Computer Science
200 University Avenue West
Waterloo, Ontario, N2L 3G1, Canada
ssrao@monod.uwaterloo.ca

Kay A. Salzwedel
Universität Paderborn
Heinz Nixdorf Institut
Fürstenallee 11
33102 Paderborn, Germany
nkz@upb.de

Peter Sanders
Max-Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
sanders@mpi-sb.mpg.de

Martin Schmollinger
Universität Tübingen
Wilhelm-Schickard Institut
für Informatik, Sand 14
72076 Tübingen, Germany
martin.schmollinger
@informatik.uni-tuebingen.de

Laura Toma
Duke University
Department of Computer Science
Durham, NC 27708, USA
laura@cs.duke.edu

Jan Vahrenhold
Westfälische Wilhelms-Universität
Institut für Informatik
Einsteinstr. 62
48149 Münster, Germany
jan@math.uni-muenster.de

Christian Weiß
Technische Universität München
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Boltzmannstr. 3
85748 München, Germany
weissc@in.tum.de

Norbert Zeh
Duke University
Department of Computer Science
Durham, NC 27708, USA
nzeh@cs.duke.edu

1. Memory Hierarchies — Models and Lower Bounds

Peter Sanders*

The purpose of this introductory chapter is twofold. On the one hand, it serves the rather prosaic purpose of introducing the basic models and notations used in the subsequent chapters. On the other hand, it explains why these simple abstract models can be used to develop better algorithms for complex real world hardware.

Section 1.1 starts with a basic motivation for memory hierarchies and Section 1.2 gives a glimpse on their current and future technological realizations. More theoretically inclined readers can skip or skim this section and directly proceed to the introduction of the much simpler abstract models in Section 1.3. Then we have all the terminology in place to explain the guiding principles behind algorithm design for memory hierarchies in Section 1.4. A further issue permeating most external memory algorithms is the existence of fundamental lower bounds on I/O complexity described in Section 1.5. Less theoretically inclined readers can skip the proofs but might want to remember these bounds because they show up again and again in later chapters.

Parallelism is another important approach to high performance computing that has many interactions with memory hierarchy issues. We describe parallelism issues in subsections that can be skipped by readers only interested in sequential memory hierarchies.

1.1 Why Memory Hierarchies

There is a wide spectrum of computer applications that can make use of arbitrarily large amounts of memory. For example, consider geographic information systems. NASA measures the data volumes from satellite images in petabytes (10^{15} bytes). Similar figures are given by climate research centers and particle physicists [555].

Although it is unlikely that all these informations will be needed in a single application, we can easily come to huge data sets. For example, consider a map of the world that associates 32 bits with each square meter of a continent — something technologically quite feasible with modern satellite imagery. We would get a data set of about 600 terabytes.

Other examples of huge data sets are *data warehouses* of large companies that keep track of every single transaction, digital libraries for books, images, and movies (a single image frame of high quality movie takes several

* Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

U. Meyer et al. (Eds.): Algorithms for Memory Hierarchies, LNCS 2625, pp. 1-13, 2003.
© Springer-Verlag Berlin Heidelberg 2003

megabytes), or large scale numerical simulations. Even if the input and output of an application are small, it might be necessary to store huge intermediate data structures. For example, some of the state space search algorithms in Chapter 11 are of this type.

How should a machine for processing such large inputs look? Data should be cheap to store but we also want fast processing. Unfortunately, there are fundamental reasons why we cannot get memory that is at the same time cheap, compact, and fast. For example, no signal can propagate faster than light. Hence, given a storage technology and a desired access latency, there is only a finite amount of data reachable within this time limit. Furthermore, in a cheap and compact storage technology there is no room for wires reaching every single memory cell. It is more economical to use a small number of devices that can be moved to access a given bit.

There are several approaches to escape this so called *memory wall* problem. The simplest and most widely used compromise is a *memory hierarchy*. There are several categories of memory in a computer ranging from small and fast to large, cheap, and slow. Even in a memory hierarchy, we can process huge data sets efficiently. The reason is that although *access latencies* to the huge data sets are large, we can still achieve large bandwidths by accessing many close-by bits together and by using several memory units in parallel. Both approaches can be modeled using the same abstract view: Access to large blocks of memory is almost as fast as access to a single bit. The algorithmic challenge following from this principle is to design algorithms that perform well on systems with blocked memory access. This is the main subject of this volume.

1.2 Current Technology

Although we view memory hierarchies as something fundamental, it is instructive to look at the way memory hierarchies are currently designed and how they are expected to change in the near future. More details and explanations can be found in the still reasonably up to date textbook [392]. A valuable and up-to-date introductory source is the web page on PC Technology `http://www.pctechguide.com/`.

Currently, a high performance microprocessor has a file of *registers* that have multiple ports so that several accesses can be made in parallel. For example, twelve parallel accesses must be supported by a superscalar machine that executes up to four instructions per clock cycle each of which addresses three registers.

Since multiple ports require too much chip area per bit, the *first level (L1) cache* supports only one or two accesses per clock. Each such access already incurs a delay of a few clock cycles since additional stages of the instruction processing pipelines have to be traversed. L1 cache is usually only a few kilobytes large because a larger area would incur longer connections and

hence even larger access latencies [399]. Often there are separate L1 caches for instructions and data.

The *second level (L2) cache* is on the same chip as the first level cache but it has quite different properties. The L2 cache is as large as the technology allows because applications that fit most of their data into this cache can execute very fast. The L2 cache has access latencies around ten clock cycles. Communication between L1 and L2 cache uses block sizes of 16–32 bytes. For accessing off-chip data, larger blocks are used. For example, the Pentium 4 uses 128 byte blocks [399].

Some processors have a *third level (L3) cache* that is on a separate set of chips. This cache is made out of fast static[1] RAM cells. The L3 cache can be very large in principle, but this is not always cost effective because static RAMs are rather expensive.

The *main memory* is made out of high density cheap dynamic RAM cells. Since the access speeds of dynamic RAMs have lagged behind processor speeds, dynamic RAMs have developed into devices optimized for block access. For example, RAMBUS RDRAM[2] chips allow blocks of up to 16 bytes to be accessed in only twice the time to access a single byte.

The programmer is not required to know about the details of the hierarchy between caches and main memory. The hardware cuts the main memory into blocks of fixed size and automatically maps a subset of the memory blocks to L3 cache. Furthermore, it automatically maps a subset of the blocks in L3 cache to L2 cache and from L2 cache to L1 cache. Although this automatic cache administration is convenient and often works well, one is up to unpleasant surprises. In Chapter 8 we will see that sometimes a careful manual mapping of data to the memory hierarchy would work much better.

The backbone of current data storage are magnetic *hard disks* because they offer cheap non volatile memory [643]. In the last years, extremely high densities have been achieved for magnetic surfaces that allow several gigabytes to be stored on the area of a postage stamp. The data is accessed by tiny magnetic devices that hover as low as 20 nm over the surface of the rotating disk. It takes very long to move the access head to a particular track of the disk and to wait until the disk rotates into the correct position. With up to 10 ms, disk access can be 10^7 times slower than an access to a register. However, once the head starts reading or writing, data can be transferred at a rate of about 50 megabytes per second. Hence, accessing hundreds of KB takes only about twice as long as accessing a single byte. Clearly, it makes sense to process data in large chunks.

Hard disks are also used as a way to virtually enlarge the main memory. Logical blocks that are currently not in use are swapped to disk. This mechanism is partially supported by the processor hardware that is able to

[1] Static RAM needs six transistors per bit which makes it more area consuming but faster than *dynamic* RAM that needs only one transistor per bit.

[2] `http://www.rambus.com`

automatically translate between logical memory addresses and physical memory addresses. This translation uses yet another small cache, the *translation lookaside buffer (TLB)*

There is a final level of memory hierarchy used for backups and archiving of data. Magnetic tapes and optical disks allow even cheaper storage of data but have a very high access latency ranging from seconds to minutes because the media have to be retrieved from a shelf and mounted on some access device.

Current and Future Developments

There are too many possible developments to explain or even perceive all of them in detail but a few basic trends should be noted. The memory hierarchy might become even deeper. Third level caches will become more common. Intel has even integrated it on the Itanium 2 processor. In such a system, an off-chip 4th level cache makes sense. There is also a growing gap between the access latencies and capacities of disks and main memory. Therefore, magnetic storage devices with smaller capacity but also lower access latency have been proposed [669].

While storage density in CMOS-RAMs and magnetic disks will keep increasing for quite some time, it is conceivable that different technologies will get their chance in a longer time frame. There are some ideas available that would allow memory cells consisting of single molecules [780]. Furthermore, even with current densities, astronomically large amounts of data could be stored using three-dimensional storage devices. The main difficulty is how to write and read such memories. One approach uses holographic images storing large blocks of data in small three-dimensional regions of a transparent material [716].

Regardless of the technology, it seems likely that block-wise access and the use of parallelism will remain necessary to achieve high performance processing of large volumes of data.

Parallelism

A more radical change in the model is explicit parallel processing. Although this idea is not so new, there are several reasons why it might have increased impact in the near future. Microprocessors like the Intel Xeon first delivered in 2002 have multiple register sets and are able to execute a corresponding number of threads of activity in parallel. These threads share the same execution pipeline. Their accumulated performance can be significantly higher than the performance of a single thread with exclusive access to the processing resources. One main reason is that while one thread is waiting for a memory access to finish, another thread can use the processor. Parallelism spreads in many other respects. Several processors on the same chip can share

a main memory and a second level cache. The IBM Power 4 processor already implements this technology. Several processors on different chips can share main memory. Several processor boards can share the same network of disks. Servers usually have many disk drives. In such systems, it becomes more and more important that memory devices on all levels of the memory hierarchy can work on multiple memory accesses in parallel.

On parallel machines, some levels of the memory hierarchy may be shared whereas others are distributed between the processors. Local caches may hold copies of shared or remote data. Thus, a read access to shared data may be as fast as a local access. However, writing shared data invalidates all the copies that are not in the cache of the writing processor. This can cause severe overhead for sending the invalidations and for reloading the data at subsequent remote accesses.

1.3 Modeling

We have seen that real memory hierarchies are very complex. We have multiple levels, all with their own idiosyncrasies. Hardware caches have replacement strategies that vary between simplistic and strange [294], disks have position dependent access delays, etc. It might seem that the best models are those that are as accurate as possible. However, for algorithm design, this leads the wrong way. Complicated models make algorithms difficult to design and analyze. Even if we overcome these differences, it would be very difficult to interpret the results because complicated models have a lot of parameters that vary from machine to machine.

Attractive models for algorithm design are very simple, so that it is easy to develop algorithms. They have few parameters so that it is easy to compare the performance of algorithms. The main issue in model design is to find simple models that grasp the essence of the real situation so that algorithms that are good in the model are also good in reality.

In this volume, we build on the most widely used nonhierarchical model. In the *random access machine (RAM) model* or *von Neumann model* [579], we have a "sufficiently" large uniform memory storing *words* of size $\mathcal{O}(\log n)$ bits where n is the size of our input. Accessing any word in memory takes constant time. Arithmetics and bitwise operations with words can be performed in constant time. For numerical and geometric algorithms, it is sometimes also assumed that words can represent real numbers accurately. Storage consumption is measured in words if not otherwise mentioned.

Most chapters of this volume use a minimalistic extension that we will simply call the *external memory model*. We use the notation introduced by Aggarwal, Vitter, and Shriver [17, 755]. Processing works almost as in the RAM model, except that there are only M words of *internal memory* that can be accessed quickly. The remaining memory can only be accessed using *I/Os* that move B contiguous words between internal and *external memory*.

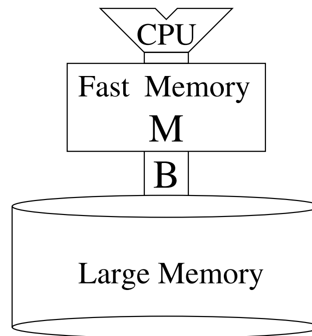


Fig. 1.1. The external memory model.

Figure 1.1 depicts this arrangement. To analyze an external memory algorithm, we count the number of I/Os needed in addition to the time that would be needed on a RAM machine.

Why is such a simple model adequate to describe something as complex as memory hierarchies? The easiest justification would be to lean on authority. Hundreds of papers using this model have been published, many of them in top conferences and journals. Many external memory algorithms developed are successfully used in practice. Vitter [754] gives an extensive overview. But why is this model so successful? Although the word "I/O" suggests that external memory should be identified with disk memory, we are free to choose any two levels of the memory hierarchy for internal and external memory in the model. Inaccuracies of the model are usually limited by reasonable constant factors. This claim needs further explanation. The main problem with hardware caches is that they use a fixed simplistic strategy for deciding which blocks are kept whereas the external memory model gives the programmer full control over the content of internal memory. Although this difference can have devastating effects, it rarely happens in practice. Mehlhorn and Sanders [543] give an explanation of this effect for a large class of cache access patterns. Sen and Chatterjee [685] and Frigo et al. [321] have observed that in principle we can even circumvent hardware replacement schemes and take explicit control of cache content.

Hard disks are even more complicated than caches [643] but again inaccuracies of the external memory model are not as big as one might think: Disks have their own local caches. But these are so small that for algorithms that process really large data sets they do not make a big difference. Roughly speaking, the disk access time consists of a latency needed to move the disk head to the appropriate position and a transfer time that is proportional to the amount of data transmitted. We are more or less free to choose this amount of data and hence it is not accurate to only count the number of accesses. However, if we fix the block size so that the transfer time is about the same as the latency, we only make a small error. Let us explain this for the (oversimplified) case that time $t_0 + B$ is needed to access B words of data.

Then a good choice of the block size is $B = t_0$. When we access less data we are at most a factor two off by accessing an entire block of size B. When we access $L > B$ words, we are at most a factor two off by counting $\lceil L/B \rceil$ block I/Os.

In reality, the access latency depends on the current position of the disk mechanism and on the position of the block to be accessed on the disk. Although exploiting this effect can make a big difference, programs that optimize access latencies are rare since the details depend on the actual disk used and are usually not published by the disk vendors. If other applications or the operating system make additional unpredictable accesses to the same disk, even sophisticated optimizations can be in vain. In summary, by picking an appropriate block size, we can model the most important aspects of disk drives.

Parallelism

Although we mostly use the sequential variant of the external memory model, it also has an option to express parallelism. External memory is partitioned into D parts (e.g. disks) so that in each I/O step, one block can be accessed on *each* of the parts.

With respect to parallel disks, the model of Vitter and Shriver [755] deviates from an earlier model by Aggarwal and Vitter [17] where D *arbitrary* blocks can be accessed in parallel. A hardware realization could have D reading/writing devices that access a single disk or a D-ported memory. This model is more powerful because algorithms need not care about the mapping of data to disks. However, there are efficient (randomized) algorithms for emulating the Aggarwal-Vitter model on the Vitter-Shriver model [656]. Hence, one approach to developing parallel disk external memory algorithms is to start with an algorithm for the Aggarwal-Vitter model and then add an appropriate load balancing algorithm (also called *declustering*).

Vitter and Shriver also make provisions for parallel processing. There are P identical processors that can work in parallel. Each has fast memory M/P and is equipped with D/P disks. In the external memory model there are no additional parameters expressing the communication capabilities of the processors. Although this is an oversimplification, this is already enough to distinguish many algorithms with respect to their ability to be executed on parallel machines. The model seems suitable for parallel machines with shared memory.

For discussing parallel external memory on machines with distributed memory we need a model for communication cost. The *BSP model* [742] that is widely accepted for parallel (internal) processing fits well here: The P processors work in *supersteps*. During a superstep, the processors can perform local communications and post messages to other processors to the communication subsystem. At the end of a superstep, all processors synchronize and exchange all the messages that have been posted during the superstep. This

synchronous communication takes time $\ell + gh$ where ℓ is the *latency*, g the *gap* and h the maximum number of words a processor sends or receives in this communication phase. The parameter ℓ models the overhead for synchronizing the processors and the latency of messages traveling through the network. If we assume our unit of time to be the time needed to execute one instruction, the parameter g is the ratio between communication speed and computation speed.

1.3.1 More Models

In Chapter 8 we will see more refined models for the fastest levels of the memory hierarchy, including replacements strategies used by the hardware and the role of the TLB. Chapter 10 contributes additional practical examples from numeric computing. Chapter 15 will explain parallel models in more detail. In particular, we will see models that take multiple levels of hierarchy into account.

There are also alternative models for the simple sequential memory hierarchy. For example, instead of counting block I/Os with respect to a block size B, we could allow variable block sizes and count the *number* of I/Os k and the total I/O *volume* h. The total I/O cost could then be accounted as $\ell_{\mathrm{I/O}} k + g_{\mathrm{I/O}} v$ where — in analogy to the BSP model — $\ell_{\mathrm{I/O}}$ stands for the I/O latency and $g_{\mathrm{I/O}}$ for the ratio between I/O speed and computation speed. This model is largely equivalent to the block based model but it might be more elegant when used together with the BSP model and it is more adequate to explain differences between algorithms with regular and irregular access patterns [227].

Another interesting variant is the *cache oblivious* model discussed in Chapter 9. This model is identical to the external memory model except that the algorithm is not told the values of B and M. The consequence of this seemingly innocent variant is that an I/O efficient cache oblivious algorithm works well not only on any machine but also on all levels of the memory hierarchy at the same time. Cache oblivious algorithms can be very simple, i.e., we do not need to know B and M to scan an array. But even cache oblivious sorting is quite difficult.

Finally, there are interesting approaches to *eliminate* memory hierarchies. Blocked access is only one way to hide access latency. Another approach is *pipelining* where many independent accesses are executed in parallel. This approach is more powerful but also more difficult to support in hardware. Vector computers such as the NEC SX-6 support pipelined memory access even to nonadjacent cells at full memory bandwidth. Several experimental machines [2, 38] use massive pipelined memory access by the hardware to run many parallel threads on a single processor. While one thread waits for a memory access, the other threads can do useful work. Modern mainstream processors also support pipelined memory access to a certain extend [399].

1.4 Issues in External Memory Algorithm Design

Before we look at particular algorithms in the rest of this volume, let us first discuss the goals we should achieve by an external memory algorithm. Ideally, the user should not notice the difference between external memory and internal memory at all, i.e., the program should run as fast as if all the memory would be internal memory. The following principles help:

Internal efficiency: The internal work done by the algorithm should be comparable to the best internal memory algorithms.

Spatial locality: When a block is accessed, it should contain as much useful data as possible.

Temporal locality: Once data is in the internal memory, as much useful work as possible should be done on it before it is written back to external memory.

Which of these criteria is most important, depends a lot on the application and on the hardware used. As usual in computer science, the overall performance is mostly determined by the weakest link. Let us consider a prototypical scenario. Assume we have a good internal memory algorithm for some application. Now it turns out that we want to run it on much larger inputs and internal memory will not suffice any more. The first try could be to ignore this problem and see how the virtual memory capability of the operating system deals with it. When this works, we are exceptionally lucky. If we see very bad performance, this usually means that the existing algorithm has poor locality. We may then apply the algorithmic techniques developed in this volume to improve locality.

Several outcomes are possible. It may be that despite our effort, locality remains the limiting factor. When discussing further improvements we will then focus on locality and might even accept an increase of internal work.

But we should keep in mind that many algorithms do some useful work for every word accessed, i.e., locality is quite good. If the application nevertheless remains *I/O-bound*, this means that the I/O bandwidth of our system is low. This is a common observation when researchers run their external memory algorithms on workstations with a single disk and I/O interfaces not build for high performance I/O. However, we should expect that serious applications of external memory algorithms will run on hardware and software build for high I/O performance. Let us consider a machine recently configured by Roman Dementiev and the author as an example. The parts for this system cost about 3000 Euro in July 2002, i.e., the price is in the range of an ordinary workstation. The STREAM[3] benchmark achieves a main memory bandwidth of 1445MB/s on one of two 2.0 GHz Intel Xeon processors. Using eight disks and four IDE controllers, we achieve an I/O bandwidth of up to 375 MB/s, i.e., the bandwidth gap between main memory and disks is not very large.

[3] `http://www.streambench.org/`

For example, our first implementation of external memory sorting on this machine used internal quicksort as a subroutine. For more than two disks this internal sorting was the main bottleneck. Hence, internal efficiency is a really important aspect of good external memory algorithms.

Parallelism

In parallel models, internal efficiency and locality is as important as in the sequential case. In particular, temporal and spatial locality with respect to the local memory of a processor is an issue.

An new issue is *load balancing* or *declustering* . All disks should access useful data in most parallel I/O steps. All processors should have about the same amount of work during a superstep in the BSP model, and no processor should have to send or receive too much data at the end of a superstep.

When programming shared memory machines, the caching policies described above must be taken into account. *True sharing* occurs when several processors write to the same memory location. Such writes are expensive since they amount to invalidation and reloading of entire cache blocks by all other processors reading this location. Hence, parallel algorithms should avoid frequent write accesses to shared data. Moreover, even write accesses to different memory cells might lead to the same sharing effect if the cells are located on the same block of memory. This phenomenon is called *false sharing*. Chapter 16 studies true and false sharing in detail using sorting as an example.

1.5 Lower Bounds

A large number of external memory algorithms can be assembled from the three ingredients scanning, sorting, and searching. There are essentially matching upper and lower bounds for the number of I/Os needed to perform these operations:

Scanning: Look at the input once in the order it is stored. If N is the amount of data to be inspected, we obviously need

$$\mathrm{scan}(N) = \Theta(N/B) \text{ I/Os.} \tag{1.1}$$

Permuting and Sorting: Too often, the data is not arranged in a way that scanning helps. Then we can rearrange the data into an order where scanning is useful. When we already know where to place each elements, this means *permuting* the data. When the permutation is defined implicitly via a total ordering “$<$” of the elements, we have to sort with respect to “$<$”. Chapter 3 gives an upper bound of

$$\mathrm{sort}(N) = \Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) \text{ I/Os} \tag{1.2}$$

for sorting. In Section 1.5.1, we will see an almost identical lower bound for permuting that is also a lower bound for the more difficult problem of sorting.
Searching: Any pointer based data structure indexing N elements needs access time

$$\text{search}(N) = \Omega\left(\log_B N/M\right)\ \text{I/Os.} \tag{1.3}$$

This lower bound is explained in Section 1.5.2. In Chapter 2 we see a matching upper bound for the simple case of a linear order. High dimensional problems such as the geometric data structures explained in Chapter 6 can be more difficult.

Arge and Bro Miltersen [59] give a more detailed account of lower bounds for external memory algorithms.

1.5.1 Permuting and Sorting

We analyse the following problem. How many I/O operations are necessary to generate a permutation of the input? A lower bound on permuting implies a lower bound for sorting because for every permutation of a set of elements, there is a set of keys that forces sorting to produce this permutation. The lower bound was established in a seminal paper by Aggarwal and Vitter [17]. Here we report a simplified proof based on unpublished lecture notes by Albers, Crauser, and Mehlhorn [24].

To establish a lower bound, we need to specify precisely what a permutation algorithm can do. We make some restrictions but most of them can be lifted without changing the lower bound significantly. We view the internal memory as a bag being able to hold up to M elements. External memory is an array of elements. Reading and writing external memory is always *aligned* to block boundaries, i.e., if the cells of external memory are numbered, access is always to cells $i, \ldots, i+B-1$ such that i is a multiple of B. At the beginning, the first N/B blocks of external memory contain the input. The internal memory and the remaining external memory contain no elements. At the end, the output is again in the first N/B blocks of the external memory. We view our elements as abstract objects, i.e., the only operation available on them is to move them around. They cannot be duplicated, split, or modified in any way. A read step moves B elements from a block of external memory to internal memory. A write step moves any B elements from internal memory into a block of external memory. In this model, the following theorem holds:

Theorem 1.1. *Permuting N elements takes at least*

$$t \geq 2\ \frac{N}{B} \cdot \frac{\log(N/eB)}{\log(eM/B) + 2\log(N/B)/B}\ \textit{I/Os.}$$

For $N = \mathcal{O}\big((eM/B)^{B/2}\big)$ the term $\log(eM/B)$ dominates the denominator and we get a lower bound for sorting of

$$2\frac{N}{B} \cdot \frac{\log(N/eB)}{\mathcal{O}(\log(eM/B))} = \Omega\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right) \ .$$

which is the same as the upper bound from Chapter 3.

The basic approach for establishing Theorem 1.1 is simple. We find an upper bound c_t for the number of different permutations generated after t I/O steps looking at all possible sequences of t I/O steps. Since there are $N!$ possible permutations of N elements, t must be large enough such that $c_t \geq N!$ because otherwise there are permutations that cannot be generated using t I/Os. Solving for t yields the desired lower bound.

A state of the algorithm can be described abstractly as follows:

1. the set of elements in the main memory;
2. the set of elements in each nonempty block of external memory;
3. the permutation in which the elements in each nonempty block of external memory are stored.

We call two states *equivalent* if they agree in the first two components (they may differ in the third).

In the final state, the elements are stored in N/B blocks of B elements each. Each equivalence class of final states therefore consists of $(B!)^{N/B}$ states. Hence, it suffices for our lower bound to find out when the number of equivalence classes of final states C_t reachable after t I/Os exceeds $N!/(B!)^{N/B}$.

We estimate C_t inductively. Clearly $C_0 = 1$.

Lemma 1.2. $C_{t+1} \leq \begin{cases} C_t N/B & \textit{if the I/O-operation is a read} \\ C_t N/B \cdot \binom{M}{B} & \textit{if the I/O-operation is a write.} \end{cases}$

Proof. A read specifies which out of N/B nonempty blocks is to be read. A write additionally specifies which elements are to be written and in which permutation. If i elements are written, there are $\binom{M}{i} \leq \binom{M}{B}$ choices for the elements to be written and their permutation is irrelevant as far as equivalence of states is concerned. The inequality $\binom{M}{i} \leq \binom{M}{B}$ assumes that $B \leq M/2$. ■

Lemma 1.3. *In any algorithm that produces a permutation in our model, the number of reads equals the number of writes.*

Proof. A read increments the number of empty blocks. A write decrements the number of empty blocks. At the beginning and at the end there are exactly N/B nonempty blocks. Hence, the number of increases equals the number of decreases. ■

Combining Lemmas 1.2 and 1.3 we see that for even t,

$$\frac{N!}{(B!)^{N/B}} \leq C_t \leq \left(\frac{N}{B}\right)^t \cdot \binom{M}{B}^{t/2} \tag{1.4}$$

We can simplify this relation using the the well-known bounds $(m/e)^m \leq m! \leq m^m$. We get $\binom{M}{B} \leq M^B/B! \leq (eM/B)^B$ and $N!/(B!)^{N/B} \geq (N/e)^N/B^N$. Relation 1.4 therefore implies

$$\left(\frac{N}{B}\right)^t \cdot \left(\frac{eM}{B}\right)^{Bt/2} \geq \left(\frac{N}{eB}\right)^N,$$

or, after taking logarithms and solving for t,

$$t \cdot \left(2\log\left(\frac{N}{B}\right) + B\log\left(\frac{eM}{B}\right)\right) \geq 2N\log\left(\frac{N}{eB}\right) \text{ or}$$

$$t \geq 2\,\frac{N}{B} \cdot \frac{\log(N/eB)}{\log(eM/B) + 2\log(N/B)/B}.$$

■

1.5.2 Pointer Based Searching

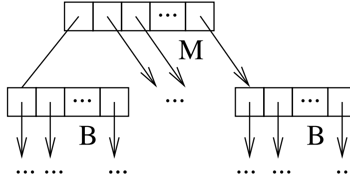


Fig. 1.2. Pointer based searching.

Consider a data structure storing a set of N elements in external memory. Access to blocks of external memory is *pointer based*, i.e., we are only allowed to access a block i if its address is actually stored somewhere. We play a similar game as for the sorting bound and count the number of different blocks C_t that can be accessed after i I/O operations. This count has to exceed N/B to make all elements accessible. Initially, the fast memory could be full of pointers so that we have $C_0 = M$. Each additional block read gives us a factor B more possibilities. Hence, $C_t = MB^t$. Figure 1.2 illustrates this situation. Solving $C_t \geq N/B$ yields $N \geq \log_B \frac{N}{MB}$ since we need an additional access for actually retrieving an element we get a lower bound of $\log_B N/M$ I/Os for being able to reach each of the N elements. ■

MORGAN & CLAYPOOL PUBLISHERS

Shared-Memory Synchronization

Michael L. Scott

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Mark D. Hill, *Series Editor*

Shared-Memory Synchronization

Synthesis Lectures on Computer Architecture

Editor
Mark D. Hill, ***University of Wisconsin, Madison***

Synthesis Lectures on Computer Architecture publishes 50- to 100-page publications on topics pertaining to the science and art of designing, analyzing, selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. The scope will largely follow the purview of premier computer architecture conferences, such as ISCA, HPCA, MICRO, and ASPLOS.

Shared-Memory Synchronization
Michael L. Scott
2013

Resilient Architecture Design for Voltage Variation
Vijay Janapa Reddi and Meeta Sharma Gupta
2013

Multithreading Architecture
Mario Nemirovsky and Dean M. Tullsen
2013

Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)
Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Jee Choi, and Wen-mei Hwu
2012

Automatic Parallelization: An Overview of Fundamental Compiler Techniques
Samuel P. Midkiff
2012

Phase Change Memory: From Devices to Systems
Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran
2011

Multi-Core Cache Hierarchies
Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar
2011

A Primer on Memory Consistency and Cache Coherence
Daniel J. Sorin, Mark D. Hill, and David A. Wood
2011

Dynamic Binary Modification: Tools, Techniques, and Applications
Kim Hazelwood
2011

Quantum Computing for Computer Architects, Second Edition
Tzvetan S. Metodi, Arvin I. Faruque, and Frederic T. Chong
2011

High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities
Dennis Abts and John Kim
2011

Processor Microarchitecture: An Implementation Perspective
Antonio González, Fernando Latorre, and Grigorios Magklis
2010

Transactional Memory, 2nd edition
Tim Harris, James Larus, and Ravi Rajwar
2010

Computer Architecture Performance Evaluation Methods
Lieven Eeckhout
2010

Introduction to Reconfigurable Supercomputing
Marco Lanzagorta, Stephen Bique, and Robert Rosenberg
2009

On-Chip Networks
Natalie Enright Jerger and Li-Shiuan Peh
2009

The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It
Bruce Jacob
2009

Fault Tolerant Computer Architecture
Daniel J. Sorin
2009

The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines
Luiz André Barroso and Urs Hölzle
2009

Computer Architecture Techniques for Power-Efficiency
Stefanos Kaxiras and Margaret Martonosi
2008

Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency
Kunle Olukotun, Lance Hammond, and James Laudon
2007

Transactional Memory
James R. Larus and Ravi Rajwar
2006

Quantum Computing for Computer Architects
Tzvetan S. Metodi and Frederic T. Chong
2006

Copyright © 2013 by Morgan & Claypool

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopy, recording, or any other except for brief quotations in printed reviews, without the prior permission of the publisher.

Shared-Memory Synchronization

Michael L. Scott

www.morganclaypool.com

ISBN: 9781608459568 paperback
ISBN: 9781608459575 ebook

DOI 10.2200/S00499ED1V01Y201304CAC023

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Lecture #23
Series Editor: Mark D. Hill, *University of Wisconsin, Madison*
Series ISSN
Synthesis Lectures on Computer Architecture
Print 1935-3235 Electronic 1935-3243

Shared-Memory Synchronization

Michael L. Scott
University of Rochester

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #23

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Since the advent of time sharing in the 1960s, designers of concurrent and parallel systems have needed to synchronize the activities of threads of control that share data structures in memory. In recent years, the study of synchronization has gained new urgency with the proliferation of multicore processors, on which even relatively simple user-level programs must frequently run in parallel.

This lecture offers a comprehensive survey of shared-memory synchronization, with an emphasis on "systems-level" issues. It includes sufficient coverage of architectural details to understand correctness and performance on modern multicore machines, and sufficient coverage of higher-level issues to understand how synchronization is embedded in modern programming languages.

The primary intended audience is "systems programmers"—the authors of operating systems, library packages, language run-time systems, concurrent data structures, and server and utility programs. Much of the discussion should also be of interest to application programmers who want to make good use of the synchronization mechanisms available to them, and to computer architects who want to understand the ramifications of their design decisions on systems-level code.

KEYWORDS

atomicity, barriers, busy-waiting, conditions, locality, locking, memory models, monitors, multiprocessor architecture, nonblocking algorithms, scheduling, semaphores, synchronization, transactional memory

To Kelly, my wife and partner
of more than 30 years.

Contents

Preface . . . . **xv**

Acknowledgments . . . . **xvii**

1 Introduction **1**

1.1 Atomicity 3
1.2 Condition Synchronization 5
1.3 Spinning vs. Blocking 6
1.4 Safety and Liveness 7

2 Architectural Background **9**

2.1 Cores and Caches: Basic Shared-Memory Architecture 9
2.1.1 Temporal and Spatial Locality 11
2.1.2 Cache Coherence 12
2.1.3 Processor (Core) Locality 13
2.2 Memory Consistency 14
2.2.1 Sources of Inconsistency 14
2.2.2 Special Instructions to Order Memory Access 15
2.2.3 Example Architectures 19
2.3 Atomic Primitives 21
2.3.1 The ABA Problem 23
2.3.2 Other Synchronization Hardware 25

3 Essential Theory **27**

3.1 Safety 27
3.1.1 Deadlock Freedom 28
3.1.2 Atomicity 29
3.2 Liveness 37
3.2.1 Nonblocking Progress 37
3.2.2 Fairness 39
3.3 The Consensus Hierarchy 41
3.4 Memory Models 42

3.4.1 Formal Framework . . . 43
3.4.2 Data Races . . . 45
3.4.3 Real-World Models . . . 47

4 Practical Spin Locks . . . 49

4.1 Classical load-store-only Algorithms . . . 49
4.2 Centralized Algorithms . . . 53
4.2.1 Test_and_set Locks . . . 53
4.2.2 The Ticket Lock . . . 54
4.3 Queued Spin Locks . . . 56
4.3.1 The MCS Lock . . . 56
4.3.2 The CLH Lock . . . 59
4.3.3 Which Spin Lock Should I Use? . . . 64
4.4 Interface Extensions . . . 65
4.5 Special-Case Optimizations . . . 66
4.5.1 Locality-conscious Locking . . . 66
4.5.2 Double-checked Locking . . . 68
4.5.3 Asymmetric Locking . . . 68

5 Busy-wait Synchronization with Conditions . . . 71

5.1 Flags . . . 71
5.2 Barrier Algorithms . . . 72
5.2.1 The Sense-Reversing Centralized Barrier . . . 73
5.2.2 Software Combining . . . 74
5.2.3 The Dissemination Barrier . . . 75
5.2.4 Non-combining Tree Barriers . . . 76
5.2.5 Which Barrier Should I Use? . . . 79
5.3 Barrier Extensions . . . 80
5.3.1 Fuzzy Barriers . . . 80
5.3.2 Adaptive Barriers . . . 81
5.3.3 Barrier-like Constructs . . . 83
5.4 Combining as a General Technique . . . 84

6 Read-mostly Atomicity . . . 87

6.1 Reader-Writer Locks . . . 87
6.1.1 Centralized Algorithms . . . 88
6.1.2 Queued Reader-Writer Locks . . . 89

6.2 Sequence Locks 94
6.3 Read-Copy Update 97

7 Synchronization and Scheduling 103

7.1 Scheduling 103
7.2 Semaphores 105
7.3 Monitors 108
7.3.1 Hoare Monitors 109
7.3.2 Signal Semantics 111
7.3.3 Nested Monitor Calls 112
7.3.4 Java Monitors 112
7.4 Other Language Mechanisms 114
7.4.1 Conditional Critical Regions 114
7.4.2 Futures 115
7.4.3 Series-Parallel Execution 116
7.5 Kernel/User Interactions 118
7.5.1 Context Switching Overhead 118
7.5.2 Preemption and Convoys 119
7.5.3 Resource Minimization 121

8 Nonblocking Algorithms 123

8.1 Single-location Structures 123
8.2 The Michael and Scott (M&S) Queue 125
8.3 Harris and Michael (H&M) Lists 128
8.4 Hash Tables 131
8.5 Skip Lists 134
8.6 Double-ended Queues 135
8.6.1 Unbounded Lock-free Deques 136
8.6.2 Obstruction-free Bounded Deques 136
8.6.3 Work-stealing Queues 139
8.7 Dual Data Structures 140
8.8 Nonblocking Elimination 142
8.9 Universal Constructions 143

9 Transactional Memory 145

9.1 Software TM 147
9.1.1 Dimensions of the STM Design Space 148

9.1.2 Buffering of Speculative State 150
9.1.3 Access Tracking and Conflict Resolution 151
9.1.4 Validation 152
9.1.5 Contention Management 155
9.2 Hardware TM 156
9.2.1 Dimensions of the HTM Design Space 156
9.2.2 Speculative Lock Elision 160
9.2.3 Hybrid TM 163
9.3 Challenges 165
9.3.1 Semantics 165
9.3.2 Extensions 167
9.3.3 Implementation 169
9.3.4 Debugging and Performance Tuning 171

Bibliography 173

Author's Biography 203

Preface

This lecture grows out of some 25 years of experience in synchronization and concurrent data structures. Though written primarily from the perspective of systems software, it reflects my conviction that the field cannot be understood without a solid grounding in both concurrency theory and computer architecture.

Chapters 4, 5, and 7 are in some sense the heart of the lecture: they cover spin locks, busy-wait condition synchronization (barriers in particular), and scheduler-based synchronization, respectively. To set the stage for these, Chapter 2 surveys aspects of multicore and multiprocessor architecture that significantly impact the design or performance of synchronizing code, and Chapter 3 introduces formal concepts that illuminate issues of feasibility and correctness.

Chapter 6 considers atomicity mechanisms that have been optimized for the important special case in which most operations are read-only. Later, Chapter 8 provides a brief introduction to *nonblocking algorithms*, which are designed in such a way that all possible thread interleavings are correct. Chapter 9 provides a similarly brief introduction to *transactional memory*, which uses speculation to implement atomicity without (in typical cases) requiring mutual exclusion. (A full treatment of both of these topics is beyond the scope of the lecture.)

Given the volume of material, readers with limited time may wish to sample topics of particular interest. All readers, however, should make sure they are familiar with the material in Chapters 1 through 3. In my experience, practitioners often underestimate the value of formal foundations, and theoreticians are sometimes vague about the nature and impact of architectural constraints. Readers may also wish to bookmark Table 2.1 (page 19), which describes the memory model assumed by the pseudocode. Beyond that:

- Computer architects interested in the systems implications of modern multicore hardware may wish to focus on Sections 2.2–2.3.1, 3.3–3.4, 4.2–4.3, 4.5.1, 5.1–5.2, 8.1–8.3, and 9.2.
- Programmers with an interest in operating systems and run-time packages may wish to focus on Sections 2.2–2.3.1, all of Chapters 3–6, and Section 7.5.
- Authors of parallel libraries may wish to focus on Sections 2.2–2.3.1 and 5.4, plus all of Chapters 3, 7, and 8.
- Compiler writers will need to understand all of Chapters 2 and 3, plus Sections 4.5.2, 5.1, 5.3.1, 5.3.3, and 7.3–7.4.

Some readers may be surprised to find that the lecture contains no concrete performance results. This omission reflects a deliberate decision to focus on qualitative comparisons among

algorithmic alternatives. Performance is obviously of great importance in the evaluation of synchronization mechanisms and concurrent data structures (and my papers are full of hard numbers), but the constants change with time, and they depend in many cases on characteristics of the specific application, language, operating system, and hardware at hand. When relative performance is in doubt, system designers would be well advised to benchmark the alternatives in their own particular environment.

Michael L. Scott
April 2013

Acknowledgments

This lecture has benefited from the feedback of many generous colleagues. Sarita Adve, Hans Boehm, Dave Dice, Maurice Herlihy, Mark Hill, Victor Luchangco, Paul McKenney, Maged Michael, Nir Shavit, and Mike Swift all read through draft material, and made numerous helpful suggestions for improvements. I am particularly indebted to Hans for his coaching on memory consistency models and to Victor for his careful vetting of Chapter 3. (The mistakes that remain are of course my own!) My thanks as well to the students of Mark's CS 758 course in the fall of 2012, who provided additional feedback. Finally, my admiration and thanks both to Mark and to Mike Morgan for their skillful shepherding of the Synthesis series, and for convincing me to undertake the project.

Michael L. Scott
April 2013

CHAPTER 1

Introduction

In computer science, as in real life, concurrency makes it much more difficult to reason about events. In a linear sequence, if E_1 occurs before E_2, which occurs before E_3, and so on, we can reason about each event individually: E_i begins with the state of the world (or the program) after E_{i-1}, and produces some new state of the world for E_{i+1}. But if the sequence of events $\{E_i\}$ is concurrent with some other sequence $\{F_i\}$, all bets are off. The state of the world prior to E_i can now depend not only on E_{i-1} and its predecessors, but also on some prefix of $\{F_i\}$.

Consider a simple example in which two threads attempt—concurrently—to increment a shared global counter:

```
thread 1:                    thread 2:
    ctr++                        ctr++
```

On any modern computer, the increment operation (ctr++) will comprise at least three separate instruction steps: one to load ctr into a register, a second to increment the register, and a third to store the register back to memory. This gives us a pair of concurrent sequences:

```
thread 1:                    thread 2:
1:  r := ctr                 1:  r := ctr
2:  inc r                    2:  inc r
3:  ctr := r                 3:  ctr := r
```

Intuitively, if our counter is initially 0, we should like it to be 2 when both threads have completed. If each thread executes line 1 before the other executes line 3, however, then both will store a 1, and one of the increments will be "lost."

The problem here is that concurrent sequences of events can *interleave* in arbitrary ways, many of which may lead to incorrect results. In this specific example, only two of the $\binom{6}{3} = 20$ possible interleavings—the ones in which one thread completes before the other starts—will produce the result we want.

Synchronization is the art of precluding interleavings that we consider incorrect. In a distributed (i.e., message-passing) system, synchronization is subsumed in communication: if thread T_2 receives a message from T_1, then in all possible execution interleavings, all the events performed by T_1 prior to its send will occur before any of the events performed by T_2 after its receive. In a shared-memory system, however, things are not so simple. Instead of exchanging messages, threads with shared memory communicate *implicitly* through loads and stores. Implicit communication gives the programmer substantially more flexibility in algorithm design, but it requires

separate mechanisms for explicit synchronization. Those mechanisms are the subject of this lecture.

Significantly, the need for synchronization arises whenever operations are concurrent, regardless of whether they actually run in parallel. This observation dates from the earliest work in the field, led by Edsger Dijkstra [1965, 1968a, 1968b] and performed in the early 1960s. If a single processor core context switches among concurrent operations at arbitrary times, then while some interleavings of the underlying events may be less probable than they are with truly parallel execution, they are nonetheless *possible*, and a correct program must be synchronized to protect against any that would be incorrect. From the programmer's perspective, a multiprogrammed uniprocessor with preemptive scheduling is no easier to program than a multicore or multiprocessor machine.

A few languages and systems guarantee that only one thread will run at a time, and that context switches will occur only at well defined points in the code. The resulting execution model is sometimes referred to as "cooperative" multithreading. One might at first expect it to simplify synchronization, but the benefits tend not to be significant in practice. The problem is that potential context-switch points may be hidden inside library routines, or in the methods of black-box abstractions. Absent a programming model that attaches a true or false "may cause a context switch" tag to every method of every system interface, programmers must protect against unexpected interleavings by using synchronization techniques analogous to those of truly concurrent code.

Distribution

At the level of hardware devices, the distinction between shared memory and message passing disappears: we can think of a memory cell as a simple process that receives `load` and `store` messages from more complicated processes, and sends `value` and `ok` messages, respectively, in response. While theoreticians often think of things this way (the annual *PODC* [*Symposium on Principles of Distributed Computing*] and *DISC* [*International Symposium on Distributed Computing*] conferences routinely publish shared-memory algorithms), systems programmers tend to regard shared memory and message passing as fundamentally distinct. This lecture covers only the shared-memory case.

Concurrency and Parallelism

Sadly, the adjectives "concurrent" and "parallel" are used in different ways by different authors. For some authors (including the current one), two operations are *concurrent* if both have started and neither has completed; two operations are *parallel* if they may actually execute at the same time. Parallelism is thus an *implementation of* concurrency. For other authors, two operations are concurrent if there is no correct way to assign them an order in advance; they are parallel if their executions are independent of one another, so that any order is acceptable. An interactive program and its event handlers, for example, are concurrent with one another, but not parallel. For yet other authors, two operations that may run at the same time are considered concurrent (also called *task parallel*) if they execute different code; they are parallel if they execute the *same* code using different data (also called *data parallel*).

As it turns out, almost all synchronization patterns in real-world programs (i.e., all conceptually appealing constraints on acceptable execution interleaving) can be seen as instances of either *atomicity* or *condition synchronization*. Atomicity ensures that a specified sequence of instructions participates in any possible interleavings as a single, indivisible unit—that nothing else appears to occur in the middle of its execution. (Note that the very concept of interleaving is based on the assumption that underlying machine instructions are themselves atomic.) Condition synchronization ensures that a specified operation does not occur until some necessary precondition is true. Often, this precondition is the completion of some other operation in some other thread.

1.1 ATOMICITY

The example on p. 1 requires only atomicity: correct execution will be guaranteed (and incorrect interleavings avoided) if the instruction sequence corresponding to an increment operation executes as a single indivisible unit:

```
thread 1:                         thread 2:
    atomic                            atomic
        ctr++                             ctr++
```

The simplest (but not the only!) means of implementing atomicity is to force threads to execute their operations one at a time. This strategy is known as *mutual exclusion*. The code of an atomic operation that executes in mutual exclusion is called a *critical section*. Traditionally, mutual exclusion is obtained by performing acquire and release operations on an abstract data object called a *lock*:

```
lock L

thread 1:                         thread 2:
    L.acquire()                       L.acquire()
        ctr++                             ctr++
    L.release()                       L.release()
```

The acquire and release operations are assumed to have been implemented (at some lower level of abstraction) in such a way that (1) each is atomic and (2) acquire waits if the lock is currently held by some other thread.

In our simple increment example, mutual exclusion is arguably the only implementation strategy that will guarantee atomicity. In other cases, however, it may be overkill. Consider an operation that increments a specified element in an *array* of counters:

```
ctr_inc(i):
    L.acquire()
        ctr[i]++
    L.release()
```

If thread 1 calls ctr_inc(i) and thread 2 calls ctr_inc(j), we shall need mutual exclusion only if i = j. We can increase potential concurrency with a finer *granularity* of locking—for example, by

declaring a separate lock for each counter, and acquiring only the one we need. In this example, the only downside is the space consumed by the extra locks. In other cases, fine-grain locking can introduce performance or correctness problems. Consider an operation designed to move n dollars from account i to account j in a banking program. If we want to use fine-grain locking (so unrelated transfers won't exclude one another in time), we need to acquire two locks:

```
move(n, i, j):
    L[i].acquire()
    L[j].acquire()          // (there's a bug here)
        acct[i] −= n
        acct[j] += n
    L[i].release()
    L[j].release()
```

If lock acquisition and release are expensive, we shall need to consider whether the benefit of concurrency in independent operations outweighs the cost of the extra lock. More significantly, we shall need to address the possibility of *deadlock*:

```
thread 1:                           thread 2:
    move(100, 2, 3)                     move(50, 3, 2)
```

If execution proceeds more or less in lockstep, thread 1 may acquire lock 2 and thread 2 may acquire lock 3 before either attempts to acquire the other. Both may then wait forever. The simplest solution in this case is to always acquire the lower-numbered lock first. In more general cases, if may be difficult to devise a static ordering. Alternative atomicity mechanisms—in particular, *transactional memory*, which we will consider in Chapter 9—attempt to achieve the concurrency of fine-grain locking without its conceptual complexity.

From the programmer's perspective, fine-grain locking is a means of implementing atomicity for large, complex operations using smaller (possibly overlapping) critical sections. The burden of ensuring that the implementation is correct (that it does, indeed, achieve deadlock-free atomicity for the large operations) is entirely the programmer's responsibility. The appeal of transactional memory is that it raises the level of abstraction, allowing the programmer to delegate this responsibility to some underlying system.

Whether atomicity is achieved through coarse-grain locking, programmer-managed fine-grain locking, or some form of transactional memory, the intent is that atomic regions appear to be indivisible. Put another way, any realizable execution of the program—any possible interleaving of its machine instructions—must be indistinguishable from (have the same externally visible behavior as) some execution in which the instructions of each atomic operation are contiguous in time, with no other instructions interleaved among them. As we shall see in Chapter 3, there are several possible ways to formalize this requirement, most notably *linearizability* and several variants on *serializability*.

1.2 CONDITION SYNCHRONIZATION

In some cases, atomicity is not enough for correctness. Consider, for example, a program containing a *work queue*, into which "producer" threads place tasks they wish to have performed, and from which "consumer" threads remove tasks they plan to perform. To preserve the structural integrity of the queue, we shall need each insert or remove operation to execute atomically. More than this, however, we shall need to ensure that a remove operation executes only when the queue is nonempty and (if the size of the queue is bounded) an insert operation executes only when the queue is nonfull:

```
Q.insert(d):                           Q.remove():
    atomic                                 atomic
        await ¬Q.full()                        await ¬Q.empty()
        // put d in next empty slot            // return data from next full slot
```

In the synchronization literature, a concurrent queue (of whatever sort of objects) is sometimes called a *bounded buffer*; it is the canonical example of mixed atomicity and condition synchronization. As suggested by our use of the await *condition* notation above (notation we have not yet explained how to implement), the conditions in a bounded buffer can be specified at the beginning of the critical section. In other, more complex operations, a thread may need to perform nontrivial work within an atomic operation before it knows what condition(s) it needs to wait for. Since another thread will typically need to access (and modify!) some of the same data in order to make the condition true, a mid-operation wait needs to be able to "break" the atomicity of the surrounding operation in some well-defined way. In Chapter 7 we shall see that some synchronization mechanisms support only the simpler case of waiting at the beginning of a critical section; others allow conditions to appear anywhere inside.

In many programs, condition synchronization is also useful *outside* atomic operations—typically as a means of separating "phases" of computation. In the simplest case, suppose that a task to be performed in thread B cannot safely begin until some other task (data structure initialization, perhaps) has completed in thread A. Here B may spin on a Boolean *flag* variable that is initially false and that is set by A to true. In more complex cases, it is common for a program to go through a *series* of phases, each of which is internally parallel, but must complete in its entirety before the next phase can begin. Many simulations, for example, have this structure. For such programs, a *synchronization barrier*, executed by all threads at the end of every phase, ensures that all have arrived before any is allowed to depart.

It is tempting to suppose that atomicity (or mutual exclusion, at least) would be simpler to implement—or to model formally—than condition synchronization. After all, it could be thought of as a subcase: "wait until no other thread is currently in its critical section." The problem with this thinking is the scope of the condition. By standard convention, we allow conditions to consider only the values of variables, not the states of other threads. Seen in this light, atomicity is the more demanding concept: it requires agreement among *all* threads that their operations will avoid

interfering with each other. And indeed, as we shall see in Section 3.3, atomicity is more difficult to implement, in a formal, theoretical sense.

1.3 SPINNING VS. BLOCKING

Just as synchronization patterns tend to fall into two main camps (atomicity and condition synchronization), so too do their implementations: they all employ *spinning* or *blocking*. Spinning is the simpler case. For condition synchronization, it takes the form of a trivial loop:

```
while ¬condition
    // do nothing (spin)
```

For mutual exclusion, the simplest implementation employs a special hardware instruction known as test_and_set (TAS). The TAS instruction, available on almost every modern machine, sets a specified Boolean variable to true and returns the previous value. Using TAS, we can implement a trivial *spin lock*:

```
type lock = bool := false

L.acquire():                      L.release():
    while ¬TAS(&L)                    L := false
        // spin
```

Here we have equated the acquisition of L with the act of *changing* it from false to true. The acquire operation repeatedly applies TAS to the lock until it finds that the previous value was false. As we shall see in Chapter 4, the trivial test_and_set lock has several major performance problems. It is, however, correct.

The obvious objection to spinning (also known as *busy-waiting*) is that it wastes processor cycles. In a multiprogrammed system it is often preferable to *block*—to yield the processor core to some other, runnable thread. The prior thread may then be run again later—either after some suitable interval of time (at which point it will check its condition, and possibly yield, again), or at some particular time when another thread has determined that the condition is finally true.

Processes, Threads, and Tasks

Like "concurrent" and "parallel," the terms "process," "thread," and "task" are used in different ways by different authors. In the most common usage (adopted here), a *thread* is an active computation that has the potential to share variables with other, concurrent threads. A *process* is a set of threads, together with the address space and other resources (e.g., open files) that they share. A *task* is a well-defined (typically small) unit of work to be accomplished—most often the closure of a subroutine with its parameters and referencing environment. Tasks are passive entities that may be executed by threads. They are invariably implemented at user level. The reader should beware, however, that this terminology is not universal. Many papers (particularly in theory) use "process" where we use "thread." Ada uses "task" where we use "thread." Mach uses "task" where we use "process." And some systems introduce additional words—e.g., "activation," "fiber," "filament," or "hart."

The software responsible for choosing which thread to execute when is known as a *scheduler*. In many systems, scheduling occurs at two different levels. Within the operating system, a kernel-level scheduler implements (kernel-level) threads on top of some smaller number of processor cores; within the user-level run-time system, a user-level scheduler implements (user-level) threads on top of some smaller number of kernel threads. At both levels, the code that implements threads (and synchronization) may present a library-style interface, composed entirely of subroutine calls; alternatively, the language in which the kernel or application is written may provide special syntax for thread management and synchronization, implemented by the compiler.

Certain issues are unique to schedulers at different levels. The kernel-level scheduler, in particular, is responsible for protecting applications from one another, typically by running the threads of each in a different address space; the user-level scheduler, for its part, may need to address such issues as non-conventional stack layout. To a large extent, however, the kernel and runtime schedulers have similar internal structure, and both spinning and blocking may be useful at either level.

While blocking saves cycles that would otherwise be wasted on fruitless re-checks of a condition or lock, it *spends* cycles on the context switching overhead required to change the running thread. If the average time that a thread expects to wait is less than twice the context-switch time, spinning will actually be faster than blocking. It is also the obvious choice if there is only one thread per core, as is sometimes the case in embedded or high-performance systems. Finally, as we shall see in Chapter 7, blocking (otherwise known as *scheduler-based synchronization*) must be built *on top of* spinning, because the data structures used by the scheduler itself require synchronization.

1.4 SAFETY AND LIVENESS

Whether based on spinning or blocking, a correct implementation of synchronization requires both *safety* and *liveness*. Informally, safety means that bad things never happen: we never have two threads in a critical section for the same lock at the same time; we never have all of the threads in the system blocked. Liveness means that good things eventually happen: if lock L is

---

Multiple Meanings of "Blocking"

"Blocking" is another word with more than one meaning. In this chapter, we are using it in an implementation-oriented sense, as a synonym for "de-scheduling" (giving the underlying kernel thread or hardware core to another user or kernel thread). In a similar vein, it is sometimes used in a "systems" context to refer to an operation (e.g., a "blocking" I/O request) that waits for a response from some other system component. In Chapter 3, we will use it in a more formal sense, as a synonym for "unable to make forward progress on its own." To a theoretician, a thread that is spinning on a condition that must be made true by some other thread is just as "blocked" as one that has given up its kernel thread or hardware core, and will not run again until some other thread tells the scheduler to resume it. Which definition we have in mind should usually be clear from context.

---

free and at least one thread is waiting for it, some thread eventually acquires it; if queue Q is nonempty and at least one thread is waiting to remove an element, some thread eventually does.

A bit more formally, for a given program and input, running on a given system, safety properties can always be expressed as predicates P on reachable system states S—that is, $\forall S[P(S)]$. Liveness properties require at least one extra level of quantification: $\forall S[P(S) \rightarrow \exists T[Q(T)]]$, where T is a subsequent state in the *same execution* as S, and Q is some other predicate on states. From a practical perspective, liveness properties tend to be harder than safety to ensure—or even to define; from a formal perspective, they tend to be harder to prove.

Livelock freedom is one of the simplest liveness properties. It insists that threads not execute forever without making forward progress. In the context of locking, this means that if L is free and thread T has called L.acquire(), there must exist some bound on the number of instructions T can execute before *some* thread acquires L. *Starvation freedom* is stronger. Again in the context of locks, it insists that if every thread that acquires L eventually releases it, and if T has called L.acquire(), there must exist some bound on the number of instructions T can execute before acquiring L itself. Still stronger notions of *fairness* among threads can also be defined; we consider these briefly in Section 3.2.2.

Most of our discussions of correctness will focus on safety properties. Interestingly, *deadlock freedom*, which one might initially imagine to be a matter of liveness, is actually one of safety: because deadlock can be described as a predicate that takes the current system state as input, deadlock freedom simply insists that the predicate be false in all reachable states.

CHAPTER 2

Architectural Background

The correctness and performance of synchronization algorithms depend crucially on architectural details of multicore and multiprocessor machines. This chapter provides an overview of these details. It can be skimmed by those already familiar with the subject, but should probably not be skipped in its entirety: the implications of store buffers and directory-based coherence on synchronization algorithms, for example, may not be immediately obvious, and the semantics of synchronizing instructions (ordered accesses, memory fences, and *read-modify-write* instructions) may not be universally familiar.

The chapter is divided into three main sections. In the first, we consider the implications for parallel programs of caching and coherence protocols. In the second, we consider *consistency*—the degree to which accesses to different memory locations can or cannot be assumed to occur in any particular order. In the third, we survey the various read-modify-write instructions—test_and_set and its cousins—that underlie most implementations of atomicity.

2.1 CORES AND CACHES: BASIC SHARED-MEMORY ARCHITECTURE

Figures 2.1 and 2.2 depict two of the many possible configurations of processors, cores, caches, and memories in a modern parallel machine. In a so-called *symmetric* machine, all memory banks are equally distant from every processor core. Symmetric machines are sometimes said to have a *uniform memory access* (UMA) architecture. More common today are *nonuniform memory access* (NUMA) machines, in which each memory bank is associated with a processor (or in some cases with a multi-processor *node*), and can be accessed by cores of the local processor more quickly than by cores of other processors.

As feature sizes continue to shrink, the number of cores per processor can be expected to increase. As of this writing, the typical desk-side machine has 1–4 processors with 2–16 cores each. Server-class machines are architecturally similar, but with the potential for many more processors. Small machines often employ a symmetric architecture for the sake of simplicity. The physical distances in larger machines often motivate a switch to NUMA architecture, so that local memory accesses, at least, can be relatively fast.

On some machines, each core may be multithreaded—capable of executing instructions from more than one thread at a time (current per-core thread counts range from 1–8). Each core typically has a private level-1 (L1) cache, and shares a level-2 cache with other cores in its local *cluster*. Clusters on the same processor of a symmetric machine then share a common L3

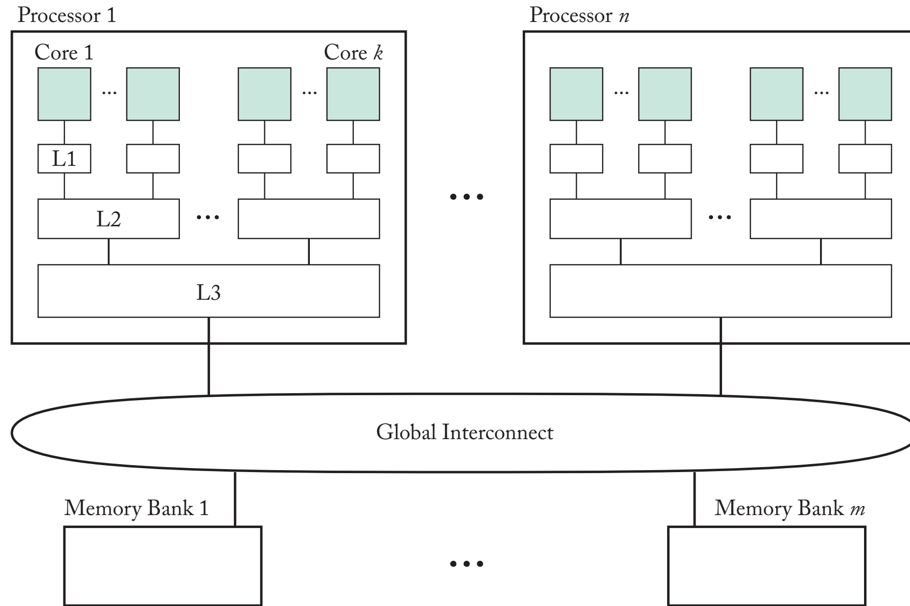


Figure 2.1: Typical symmetric (uniform memory access—UMA) machine. Numbers of components of various kinds, and degree of sharing at various levels, differs across manufacturers and models.

cache. Each cache holds a temporary copy of data currently in active use by cores above it in the hierarchy, allowing those data to be accessed more quickly than they could be if kept in memory. On a NUMA machine in which the L2 connects directly to the global interconnect, the L3 may sometimes be thought of as "belonging" to the memory.

In a machine with more than one processor, the global interconnect may have various topologies. On small machines, broadcast buses and crossbars are common; on large machines, a network of point-to-point links is more common. For synchronization purposes, broadcast has the side effect of imposing a total order on all inter-processor messages; we shall see in Section 2.2 that this simplifies the design of concurrent algorithms—synchronization algorithms in particular. Ordering is sufficiently helpful, in fact, that some large machines (notably those sold by Oracle) employ two different global networks: one for data requests, which are small, and benefit from ordering, and the other for replies, which require significantly more aggregate bandwidth, but do not need to be ordered.

As the number of cores per processor increases, on-chip interconnects—the connections among the L2 and L3 caches in particular—can be expected to take on the complexity of current global interconnects. Other forms of increased complexity are also likely, including, perhaps,

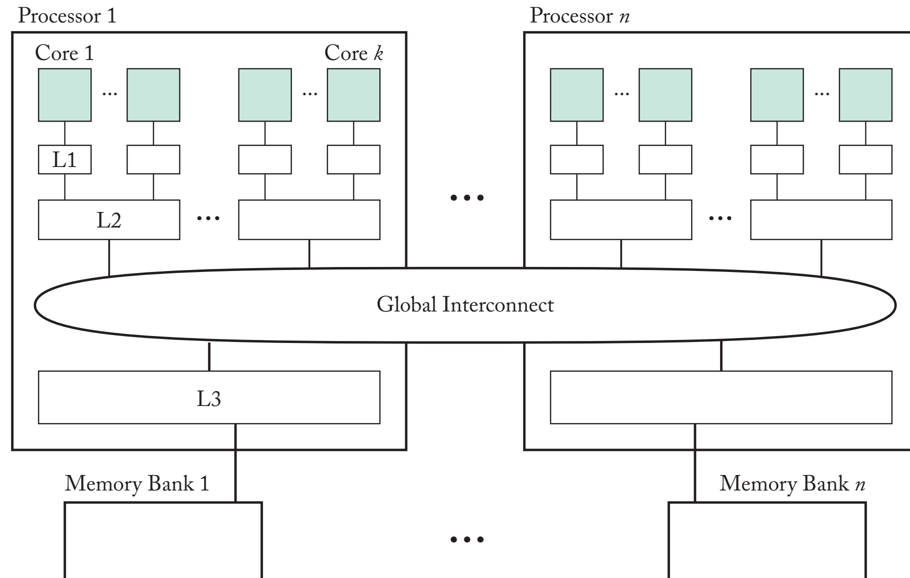


Figure 2.2: Typical nonuniform memory access (NUMA) machine. Again, numbers of components of various kinds, and degree of sharing at various levels, differs across manufacturers and models.

additional levels of caching, non-hierarchical topologies, and heterogeneous implementations or even instruction sets among cores.

The diversity of current and potential future architectures notwithstanding, multilevel caching has several important consequences for programs on almost any modern machine; we explore these in the following subsections.

2.1.1 TEMPORAL AND SPATIAL LOCALITY

In both sequential and parallel programs, performance can usually be expected to correlate with the temporal and spatial locality of memory references. If a given location l is accessed more than once by the same thread (or perhaps by different threads on the same core or cluster), performance is likely to be better if the two references are close together in time (temporal locality). The benefit stems from the fact that l is likely still to be in cache, and the second reference will be a hit instead of a miss. Similarly, if a thread accesses location l_2 shortly after l_1, performance is likely to be better if the two locations have nearby addresses (spatial locality). Here the benefit stems from the fact that l_1 and l_2 are likely to lie in the same *cache line*, so l_2 will have been loaded into cache as a side effect of loading l_1.

On current machines, cache line sizes typically vary between 32 and 512 bytes. There has been a gradual trend toward larger sizes over time. Different levels of the cache hierarchy may also use different sizes, with lines at lower levels typically being larger.

To improve temporal locality, the programmer must generally restructure algorithms, to change the order of computations. Spatial locality is often easier to improve—for example, by changing the layout of data in memory to co-locate items that are frequently accessed together, or by changing the order of traversal in multidimensional arrays. These sorts of optimizations have long been an active topic of research, even for sequential programs—see, for example, the texts of Muchnick [1997, Chap. 20] or Allen and Kennedy [2002, Chap. 9].

2.1.2 CACHE COHERENCE

On a single-core machine, there is a single cache at each level, and while a given block of memory may be present in more than one level of the memory hierarchy, one can always be sure that the version closest to the top contains up-to-date values. (With a *write-through* cache, values in lower levels of the hierarchy will never be out of date by more than some bounded amount of time. With a *write-back* cache, values in lower levels may be arbitrarily stale—but harmless, because they are hidden by copies at higher levels.)

On a shared-memory parallel system, by contrast—unless we do something special—data in upper levels of the memory hierarchy may no longer be up-to-date if they have been modified by some thread on another core. Suppose, for example, that threads on Cores 1 and k in Figure 2.1 have both been reading variable x, and each has retained a copy in its L1 cache. If the thread on Core 1 then modifies x, even if it writes its value through (or back) to memory, how do we prevent the thread on Core k from continuing to read the stale copy?

A *cache-coherent* parallel system is one in which (1) changes to data, even when cached, are guaranteed to become visible to all threads, on all cores, within a bounded (and typically small) amount of time; and (2) changes to the same location are seen in the same order by all threads. On almost all modern machines, coherence is achieved by means of an *invalidation-based cache coherence protocol*. Such a protocol, operating across the system's cache controllers, maintains the invariant that there is at most one writable copy of any given cache line anywhere in the system—and, if the number is one and not zero, there are no read-only copies.

Algorithms to maintain cache coherence are a complex topic, and the subject of ongoing research (for an overview, see the lecture of Sorin et al. [2011], or the more extensive [if dated] coverage of Culler and Singh [1998, Chaps. 5, 6, & 8]). Most protocols in use today descend from the four-state protocol of Goodman [1983]. In this protocol (using modern names for the states), each line in each cache is either *invalid* (meaning it currently holds no data block), *shared* (read-only), *exclusive* (written exactly once, and up-to-date in memory), or *modified* (written more than once, and in need of write-back).

To maintain the at-most-one-writable-copy invariant, the coherence protocol arranges, on any write to an *invalid* or *shared* line, to invalidate (evict) any copies of the block in all other caches

in the system. In a system with a broadcast-based interconnect, invalidation is straightforward. In a system with point-to-point connections, the coherence protocol typically maintains some sort of *directory* information that allows it to find all other copies of a block.

2.1.3 PROCESSOR (CORE) LOCALITY

On a single-core machine, misses occur on an initial access (a "cold miss") and as a result of limited cache capacity or associativity (a "conflict miss").[1] On a cache-coherent machine, misses may also occur because of the need to maintain coherence. Specifically, a read or write may miss because a previously cached block has been written by some other core, and has reverted to *invalid* state; a write may also miss because a previously *exclusive* or *modified* block has been read by some other core, and has reverted to *shared* state.

Absent program restructuring, coherence misses are inevitable if threads on different cores access the same datum (and at least one of them writes it) at roughly the same point in time. In addition to temporal and spatial locality, it is therefore important for parallel programs to exhibit good *thread locality*: as much possible, a given datum should be accessed by only one thread at a time.

Coherence misses may sometimes occur when threads are accessing *different* data, if those data happen to lie in the same cache block. This *false sharing* can often be eliminated—yielding a major speed improvement—if data structures are *padded* and *aligned* to occupy an integral number of cache lines. For busy-wait synchronization algorithms, it is particularly important to minimize the extent to which different threads may spin on the same location—or locations in the same cache block. Spinning with a write on a shared location—as we did in the `test_and_set` lock of Section 1.3, is particularly deadly: each such write leads to interconnect traffic proportional to the number of other spinning threads. We will consider these issues further in Chapter 4.

[1]A cache is said to be *k-way associative* if its indexing structure permits a given block to be cached in any of k distinct locations. If k is 1, the cache is said to be *direct mapped*. If a block may be held in any line, the cache is said to be *fully associative*.

No-remote-caching Multiprocessors

Most of this lecture assumes a shared-memory multiprocessor with global (distributed) cache coherence, which we have contrasted with machines in which message passing provides the only means of interprocessor communication. There is an intermediate option. Some NUMA machines (notably many of the offerings from Cray, Inc.) support a single global address space, in which any processor can access any memory location, but remote locations cannot be cached. We may refer to such a machine as a no-remote-caching (NRC-NUMA) multiprocessor. (Globally cache coherent NUMA machines are sometimes known as CC-NUMA.) Any access to a location in some other processor's memory will traverse the interconnect of an NRC-NUMA machine. Assuming the hardware implements cache coherence *within* each node—in particular, between the local processor(s) and the network interface—memory will still be globally coherent. For the sake of performance, however, system and application programmers will need to employ algorithms that minimize the number of remote references.

2.2 MEMORY CONSISTENCY

On a single-core machine, it is relatively straightforward to ensure that instructions appear to complete in execution order. Ideally, one might hope that a similar guarantee would apply to parallel machines—that memory accesses, system-wide, would appear to constitute an interleaving (in execution order) of the accesses of the various cores. For several reasons, this sort of *sequential consistency* [Lamport, 1979] imposes nontrivial constraints on performance. Most real machines implement a more *relaxed* (i.e., potentially inconsistent) memory model, in which accesses by different threads, or to different locations by the same thread, may appear to occur "out of order" from the perspective of threads on other cores. When consistency is required, programmers (or compilers) must employ special *synchronizing instructions* that are more strongly ordered than other, "ordinary" instructions, forcing the local core to wait for various classes of potentially in-flight events. Synchronizing instructions are an essential part of synchronization algorithms on any non-sequentially consistent machine.

2.2.1 SOURCES OF INCONSISTENCY

Inconsistency is a natural result of common architectural features. In an *out-of-order* processor, for example—one that can execute instructions in any order consistent with (thread-local) data dependences—a write must be held in the *reorder buffer* until all instructions that precede it in program order have completed. Likewise, since almost any modern processor can generate a burst of store instructions faster than the underlying memory system can absorb them, even writes that are logically ready to commit may need to be buffered for many cycles. The structure that holds these writes is known as a *store buffer*.

When executing a load instruction, a core checks the contents of its reorder and store buffers before forwarding a request to the memory system. This check ensures that the core always sees its own recent writes, even if they have not yet made their way to cache or memory. At the same time, a load that accesses a location that has *not* been written recently may make its way to memory before logically previous instructions that wrote to other locations. This fact is harmless on a uniprocessor, but consider the implications on a parallel machine, as shown in Figure 2.3. If the write to x is delayed in thread 1's store buffer, and the write to y is similarly delayed in thread 2's store buffer, then both threads may read a zero at line 2, suggesting that line 2 of thread 1 executes before line 1 of thread 2, and line 2 of thread 2 executes before line 1 of thread 1. When combined with program order (line 1 in each thread should execute before line 2 in the same thread), this gives us an apparent "ordering loop," which "should" be logically impossible.

Similar problems can occur deeper in the memory hierarchy. A modern machine can require several hundred cycles to service a miss that goes all the way to memory. At each step along the way (core to L1, …, L3 to bus, …) pending requests may be buffered in a queue. If multiple requests may be active simultaneously (as is common, at least, on the global interconnect), and if some requests may complete more quickly than others, then memory accesses may appear to be reordered. So long as accesses to the *same* location (by the same thread) are forced to occur in order,

```
// initially x == y == 0

thread 1:                                   thread 2:
1:    x := 1                                1:    y := 1
2:    i := y                ?               2:    j := x

// finally i == j == 0
```

Figure 2.3: An apparent ordering loop.

single-threaded code will run correctly. On a multiprocessor, however, sequential consistency may again be violated.

On a NUMA machine, or a machine with a topologically complex interconnect, differing distances among locations provide additional sources of circular ordering. If variable x in Figure 2.3 is close to thread 2 but far from thread 1, and y is close to thread 1 but far from thread 2, the reads on line 2 can easily complete before the writes on line 1, even if all accesses are inserted into the memory system in program order. With a topologically complex interconnect, the cache coherence protocol itself may introduce variable delays—e.g., to dispatch invalidation requests to the various locations that may need to change the state of a local cache line, and to collect acknowledgments. Again, these differing delays may allow line 2 of the example—in both threads—to complete before line 1.

In all the explanations of Figure 2.3, the ordering loop results from reads *bypassing* writes—executing in-order (write-then-read) from the perspective of the issuing core, but out of order (read-then-write) from the perspective of the memory system—or of threads on other cores. On NUMA or topologically complex machines, it may also be possible for reads to bypass reads, writes to bypass reads, or writes to bypass writes. Worse, circularity may arise even without bypassing—i.e., even when every thread executes its own instructions in strict program order. Consider the "independent reads of independent writes" (IRIW) example shown in Figure 2.4. If thread 1 is close to thread 2 but far from thread 3, and thread 4 is close to thread 3 but far from thread 2, the reads on line 1 in threads 2 and 3 may see the new values of x and y, while the reads on line 2 see the old. Here the problem is not bypassing, but a lack of *write atomicity*—one thread sees the value written by a store and another thread *subsequently* sees the value prior to the store. Many other examples of unintuitive behavior permitted by modern hardware can be found in the literature [Adve and Gharachorloo, 1996, Adve et al., 1999, Boehm and Adve, 2008, Manson et al., 2005].

2.2.2 SPECIAL INSTRUCTIONS TO ORDER MEMORY ACCESS

If left unaddressed, memory inconsistency can easily derail attempts at synchronization. Consider the flag-based programming idiom illustrated in Figure 2.5. If foo can never return zero, a programmer might naively expect that thread 2 will never see a divide-by-zero error at line 3. If

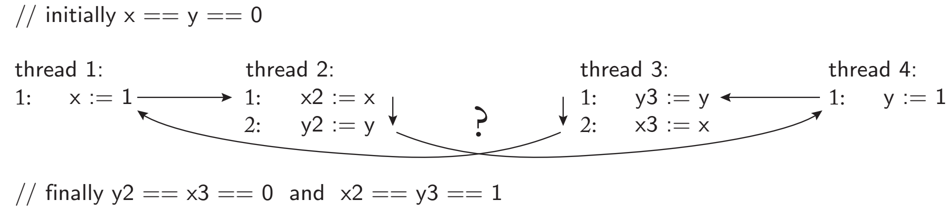


Figure 2.4: Independent reads of independent writes (IRIW). If the writes of threads 1 and 4 propagate to different places at different speeds, we can see a ordering loop even if instructions from the same thread never bypass one another.

Figure 2.5: A simple example of flag-based synchronization. To avoid a spurious error, the update to x must be visible to thread 2 before the update to f.

the write at line 2 in thread 1 can bypass the write in line 1, however, thread 2 may read x too early, and see a value of zero. Similarly, if the read of x at line 3 in thread 2 can bypass the read of f in line 1, a divide-by-zero may again occur, even if the writes in thread 1 complete in order. (While thread 2's read of x is separated from the read of f by a conditional test, the second read may still issue before the first completes, if the branch predictor guesses that the loop will never iterate.)

Compilers Also Reorder Instructions

While this chapter focuses on architectural issues, it should be noted that compilers also routinely reorder instructions. In any program not written in machine code, compilers perform a variety of optimizations in an attempt to improve performance. Simple examples include reordering computations to expose and eliminate redundancies, hoisting invariants out of loops, and "scheduling" instructions to minimize processor pipeline bubbles. Such optimizations are legal so long as they respect control and data dependences within a single thread. Like the hardware optimizations discussed in this section, compiler optimizations can lead to inconsistent behavior when more than one thread is involved. As we shall see in Section 3.4, languages designed for concurrent programming must provide a *memory model* that explains allowable behavior, and some set of primitives—typically special synchronization operations or reads and writes of special atomic variables—that serve to order accesses at the language level.

Any machine that is not sequentially consistent will provide special instructions that allow the programmer to force consistent ordering in situations in which it matters, but in which the hardware might not otherwise guarantee it. Perhaps the simplest such instruction is a synchronizing access (typically a special load or store) that is guaranteed to be both locally and globally ordered. Here "locally ordered" means that the synchronizing access will appear to occur after any preceding ordinary accesses in its own thread, and before any subsequent ordinary accesses in its thread, from the perspective of all threads. "Globally ordered" means that the synchronizing access will appear to occur in some consistent, total order with respect to all other synchronizing instructions in the program, from the perspective of all threads. (As part of global ordering, we also require that synchronizing accesses by the same thread appear to occur in program order, from the perspective of all threads.) At the hardware level, global ordering is typically achieved by means of write atomicity: the cache coherence protocol prevents a load from returning the value written by a synchronizing store until it verifies that no load elsewhere in the machine can ever again return the previous value.

To avoid the spurious error in Figure 2.5, it is sufficient (though not necessary) to use fully ordered accesses to f in both threads, thereby ensuring that thread 1's update of x happens before its update of f, and thread 2's read of x happens after it sees the change to f. Alternatively, ordering could be ensured by inserting a *full fence* instruction between lines 1 and 2 in thread 1, and between lines 2 and 3 in thread 2. A full fence doesn't read or write memory itself, but it ensures that all preceding memory accesses in its thread appear to occur before all subsequent memory accesses in its thread, from the perspective of other threads.

In Figure 2.3, circularity could be avoided by using fully ordered stores for the line-1 writes (in both threads) or by using fully ordered loads for the line-2 reads (in both threads). Alternatively, we could insert full fences between lines 1 and 2 in both threads. In Figure 2.4, however, global ordering (i.e., write atomicity) is really all that matters; to ensure it, we must use synchronizing stores in threads 1 and 4. Synchronizing loads in threads 2 and 3—or fences between the loads—will not address the problem: absent write atomicity, it is possible for thread 1's write to appear to happen before the entire set of reads—and thread 4's write after—from the perspective of thread 2, and vice versa for thread 3.

Barriers Everywhere

Fences are sometimes known as *memory barriers*. Sadly, the word *barrier* is heavily overloaded. As noted in Section 1.2 (and explored in more detail in Section 5.2), it is the name of a synchronization mechanism used to separate program phases. In the programming language community, it refers to code that must be executed when changing a pointer, in order to maintain bookkeeping information for the garbage collector. In a similar vein, it sometimes refers to code that must be executed when reading or writing a shared variable inside an atomic *transaction*, in order to detect and recover from speculation failures (we discuss this code in Chapter 9, but without referring to it as a "barrier"). The intended meaning is usually clear from context, but may be confusing to readers who are familiar with only some of the definitions.

On many machines, fully ordered synchronizing instructions turn out to be quite expensive—tens or even hundreds of cycles. Moreover, in many cases—including those described above—full ordering is more than we need for correct behavior. Architects therefore often provide a variety of weaker synchronizing instructions. These may or may not be globally ordered, and may prevent some, but not all, local bypassing. As we shall see in Section 2.2.3, the details vary greatly from one machine architecture to another. Moreover, behavior is often defined not in terms of the orderings an instruction guarantees among memory accesses, but in terms of the *re*orderings it inhibits in the processor core, the cache subsystem, or the interconnect.

Unfortunately, there is no obvious, succinct way to specify minimal ordering requirements in parallel programs. Neither synchronizing accesses nor fences, for example, allow us to order two individual accesses with respect to one another (and not with respect to anything else), if that is all that is really required. In an attempt to balance simplicity and clarity, the examples in this lecture use a notation inspired by (but simpler than) the `atomic` operations of C++'11. Using this notation, we will sometimes over-constrain our algorithms, but not egregiously.

A summary of our notation, and of the memory model behind it, can be found in Table 2.1. To specify local ordering, each synchronizing instruction admits an optional annotation of the form $P\|S$, indicating that the instruction is ordered with respect to preceding (P) and/or subsequent (S) read and write accesses in its thread ($P, S \subset \{R, W\}$). So, for example, f.store(1, W‖) might be used in Figure 2.5 at line 2 of thread 1 to order the (synchronizing) store to f after the (ordinary) write to x, and f.load(‖RW) might be used at line 1 of thread 2 to order the (synchronizing) load of f before both the (ordinary) read of x and any other subsequent reads and writes. Similarly, fence(RW‖RW) would indicate a full fence, ordered globally with respect to all other synchronizing instructions and locally with respect to all preceding and subsequent ordinary accesses in its thread.

We will assume that synchronizing instructions inhibit reordering not only by the hardware (processor, cache, or interconnect), but also by the compiler or interpreter. Compiler writers or assembly language programmers interested in porting our pseudocode to some concrete machine will need to restrict their code improvement algorithms accordingly, and issue appropriate synchronizing instructions for the hardware at hand. Beginning guidance can be found in Doug Lea's on-line "Cookbook for Compiler Writers" [2001].

To determine the need for synchronizing instructions in the code of a given synchronization algorithm, we shall need to consider both the correctness of the algorithm itself and the semantics it is intended to provide to the rest of the program. The acquire operation of Peterson's two-thread spin lock [1981], for example, employs synchronizing stores to arbitrate between competing threads, but this ordering is not enough to prevent a thread from reading or writing shared data before the lock has actually been acquired—or after it has been released. For that, one needs accesses or fences with local ‖RW and RW‖ ordering (code in Section 4.1).

Fortunately for most programmers, memory ordering details are generally of concern only to the authors of synchronization algorithms and low-level concurrent data structures, which

Table 2.1: Understanding the pseudocode

Throughout the remainder of this book, pseudocode will be set in sans serif font code (code in real programming languages will be set in `typewriter` font). We will use the term *synchronizing instruction* to refer to explicit loads and stores, fences, and atomic read-modify-write (fetch_and_Φ) operations (listed in Table 2.2). Other memory accesses will be referred to as "ordinary." We will assume the following:

coherence

All accesses (ordinary and synchronizing) to any given location appear to occur in some single, total order from the perspective of all threads.

global order

There is a global, total order on synchronizing instructions (to all locations, by all threads). Within this order, instructions issued by the same thread occur in program order.

program order

Ordinary accesses appear to occur in program order from the perspective of the issuing thread, but may bypass one another in arbitrary ways from the perspective of other threads.

local order

Ordinary accesses may also bypass synchronizing instructions, except when forbidden by an ordering annotation ({R,W}||{R,W}) on the synchronizing instruction.

values read

A read instruction will return the value written by the most recent write (to the same location) that is ordered before the read. It may also, in some cases, return the value written by an unordered write. More detail on memory models can be found in Section 3.4.

may need to be re-written (or at least re-tuned) for each target architecture. Programmers who use these algorithms correctly are then typically assured that their programs will behave as if the hardware were sequentially consistent (more on this in Section 3.4), and will port correctly across machines.

Identifying a minimal set of ordering instructions to ensure the correctness of a given algorithm on a given machine is a difficult and error-prone task. It has traditionally been performed by hand, though there has been promising recent work aimed at verifying the correctness of a given set of fences [Burckhardt et al., 2007], or even at inferring them directly [Kuperstein et al., 2010].

2.2.3 EXAMPLE ARCHITECTURES

A few multiprocessors (notably, those built around the c. 1996 MIPS R10000 processor [Yeager, 1996]) have been defined to be sequentially consistent. A few others (notably, the HP PA-RISC) have been implemented with sequential consistency, even though the documentation permitted something more relaxed. Hill [1998] has argued that the overhead of sequential consistency need not be onerous, particularly in comparison to its conceptual benefits. Most machines today, however, fall into two broad classes of more relaxed alternatives. On the SPARC, x86 (both 32- and

64-bit), and IBM z Series, reads are allowed to bypass writes, but R‖R, R‖W, and W‖W orderings are all guaranteed to be respected by the hardware, and writes are always globally ordered. Special instructions—synchronizing accesses or fences—are required only when the programmer must ensure that a write and a subsequent read complete in program order. On ARM, POWER, and IA-64 (Itanium) machines, all four combinations of local bypassing are possible: special instructions must be used whenever ordering is required. Moreover, on ARM and POWER, ordinary writes are not guaranteed to be atomic.

We will refer to hardware-level memory models in the SPARC/x86/z camp using the SPARC term *TSO* (Total Store Order). We will refer to the other machines as "more relaxed." On TSO machines, R‖R, R‖W, and W‖W annotations can all be elided from our code. On more relaxed machines, they must be implemented with appropriate machine instructions. It should be emphasized that there are significant differences among machines within a given camp—in the default ordering, the available synchronizing instructions, and the details of corner cases. A full explanation is well beyond what we can cover here. For a taste of the complexities involved, see Sewell et al.'s attempt [2010] to formalize the behavior specified informally in Intel's architecture manual [2011, Vol. 3, Sec. 8.2].

A few machines—notably the Itanium and ARM v8 (but not v7)—provide explicit synchronizing access (load and store) instructions. Most machines provide separate fence instructions, which do not, themselves, modify memory. Though less common, synchronizing accesses have two principal advantages. First, they can achieve write atomicity, thereby precluding the sort of circularity shown in Figure 2.4. Second, they allow an individual access (the load or store itself) to be ordered with respect to preceding or subsequent ordinary accesses. A fence, by contrast, must order preceding or subsequent ordinary accesses with respect to *all* reads or writes in the opposite direction.

One particular form of local ordering is worthy of special mention. As noted in Section 2.2.2, a lock acquire operation must ensure that a thread cannot read or write shared data until it has actually acquired the lock. The appropriate guarantee will typically be provided by a ‖RW load or, more conservatively, a subsequent R‖RW fence. In a similar vein, a lock release must ensure that all reads and writes within the critical section have completed before the lock is actually released. The appropriate guarantee will typically be provided by a RW‖ store or, more conservatively, a preceding RW‖W fence. These combinations are common enough that they are sometimes referred to as *acquire* and *release* orderings. They are used not only for mutual exclusion, but for most forms of condition synchronization as well. The IA-64 (Itanium), ARM v8, and several research machines—notably the Stanford Dash [Lenoski et al., 1992]—support acquire and release orderings directly in hardware. In the mutual exclusion case, particularly when using synchronizing accesses rather than fences, acquire and release orderings allow work to "migrate" into a critical section both from above (prior to the lock acquire) and from below (after the lock release). They do *not* allow work to migrate *out* of a critical section in either direction.

Table 2.2: Common atomic (read-modify-write) instructions

test_and_set
bool TAS(bool *a): atomic { t := *a; *a := true; return t }
swap
word Swap(word *a, word w): atomic { t := *a; *a := w; return t }
fetch_and_increment
int FAI(int *a): atomic { t := *a; *a := t + 1; return t }
fetch_and_add
int FAA(int *a, int n): atomic { t := *a; *a := t + n; return t }
compare_and_swap
bool CAS(word *a, word old, word new): atomic { t := (*a = old); if (t) *a := new; return t }
load_linked / store_conditional
word LL(word *a): atomic { remember a; return *a } bool SC(word *a, word w): atomic { t := (a is remembered, and has not been evicted since LL) if (t) *a := w; return t }

2.3 ATOMIC PRIMITIVES

To facilitate the construction of synchronization algorithms and concurrent data structures, most modern architectures provide instructions capable of updating (i.e., reading *and* writing) a memory location as a single atomic operation. We saw a simple example—the test_and_set instruction (TAS)—in Section 1.3. A longer list of common instructions appears in Table 2.2. Note that for each of these, when it appears in our pseudocode, we permit an optional, final argument that indicates local ordering constraints. CAS(a, old, new, W‖), for example, indicates a CAS instruction that is ordered after all preceding write accesses in its thread.

Originally introduced on mainframes of the 1960s, TAS and Swap are still available on several modern machines, among them the x86 and SPARC. FAA and FAI were introduced for "combining network" machines of the 1980s [Kruskal et al., 1988]. They are uncommon in hardware today, but frequently appear in algorithms in the literature. The semantics of TAS, Swap, FAI, and FAA should all be self-explanatory. Note that they all return the value of the target location *before* any change was made.

CAS was originally introduced in the 1973 version of the IBM 370 architecture [Brown and Smith, 1975, Gifford et al., 1987, IBM, 1975]. It is also found on modern x86, IA-64 (Ita-

nium), and SPARC machines. LL/SC was originally proposed for the S-1 AAP Multiprocessor at Lawrence Livermore National Laboratory [Jensen et al., 1987]. It is also found on modern POWER, MIPS, and ARM machines. CAS and LL/SC are *universal* primitives, in a sense we will define formally in Section 3.3. In practical terms, we can use them to build efficient simulations of arbitrary (single-word) read-modify-write (fetch_and_Φ) operations (including all the other operations in Table 2.2).

CAS takes three arguments: a memory location, an old value that is expected to occupy that location, and a new value that should be placed in the location if indeed the old value is currently there. The instruction returns a Boolean value indicating whether the replacement occurred successfully. Given CAS, fetch_and_Φ can be written as follows, for any given function Φ:

```
1:  word fetch_and_Φ(function Φ, word *w):
2:      word old, new
3:      repeat
4:          old := *w
5:          new := Φ(old)
6:      until CAS(w, old, new)
7:      return old
```

In effect, this code computes Φ(*w) *speculatively*, and then updates w atomically if its value has not changed since the speculation began. The only way the CAS can fail to perform its update (and return false at line 6) is if some other thread has recently modified w. If several threads attempt to perform a fetch_and_Φ on w simultaneously, one of them is guaranteed to succeed, and the system as a whole will make forward progress. This guarantee implies that fetch_and_Φ operations implemented with CAS are *nonblocking* (more specifically, *lock free*), a property we will consider in more detail in Section 3.2.

One problem with CAS, from an architectural point of view, is that it combines a load and a store into a single instruction, which complicates the implementation of pipelined processors. LL/SC was designed to address this problem. In the fetch_and_Φ idiom above, it replaces the load at line 4 with a special instruction that has the side effect of "tagging" the associated cache line so that the processor will "notice" any subsequent eviction of the line. A subsequent SC will then succeed only if the line is still present in the cache:

```
word fetch_and_Φ(function Φ, word *w):
    word old, new
    repeat
        old := LL(w)
        new := Φ(old)
    until SC(w, new)
    return old
```

Here any argument for forward progress requires an understanding of why SC might fail. Details vary from machine to machine. In all cases, SC is guaranteed to fail if another thread has modified

*w (the location pointed at by w) since the LL was performed. On most machines, SC will also fail if a hardware interrupt happens to arrive in the post-LL window. On some machines, it will fail if the cache suffers a capacity or conflict miss, or if the processor mispredicts a branch. To avoid deterministic, spurious failure, the programmer may need to limit (perhaps severely) the types of instructions executed between the LL and SC. If unsafe instructions are required in order to compute the function Φ, one may need a hybrid approach:

```
1:  word fetch_and_Φ(function Φ, word *w):
2:      word old, new
3:      repeat
4:          old :=*w
5:          new := Φ(old)
6:      until LL(w) = old && SC(w, new)
7:      return old
```

In effect, this code uses LL and SC at line 6 to emulate CAS.

2.3.1 THE ABA PROBLEM

While both CAS and LL/SC appear in algorithms in the literature, the former is quite a bit more common—perhaps because its semantics are self-contained, and do not depend on the implementation-oriented side effect of cache-line tagging. That said, CAS has one significant disadvantage from the programmer's point of view—a disadvantage that LL/SC avoids.

Because it chooses whether to perform its update based on the value in the target location, CAS may succeed in situations where the value has changed (say from A to B) and then changed back again (from B to A) [IBM, 1975, 1983, Treiber, 1986]. In some algorithms, such a change and restoration is harmless: it is still acceptable for the CAS to succeed. In other algorithms, incorrect behavior may result. This possibility, often referred to as the *ABA problem*, is particularly worrisome in pointer-based algorithms. Consider the following (buggy!) code to manipulate a linked-list stack:

Emulating CAS

Note that while LL/SC can be used to emulate CAS, the emulation requires a loop to deal with spurious SC failures. This issue was recognized explicitly by the designers of the C++'11 `atomic` types and operations, who introduced two variants of CAS. The `atomic_compare_exchange_strong` operation has the semantics of hardware CAS: it fails only if the expected value was not found. On an LL/SC machine, it is implemented with a loop. The `atomic_compare_exchange_weak` operation admits the possibility of spurious failure: it has the interface of CAS, but is implemented *without* a loop on an LL/SC machine.

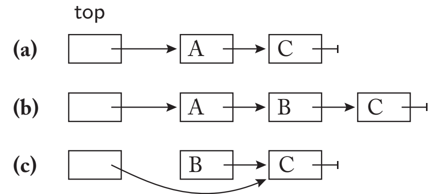


Figure 2.6: The ABA problem in a linked-list stack.

```
1: void push(node** top, node* new):
2:     node* old
3:     repeat
4:         old := *top
5:         new→next := old
6:     until CAS(top, old, new)
```

```
1: node* pop(node** top):
2:     node* old, new
3:     repeat
4:         old := *top
5:         if old = null return null
6:         new := old→next
7:     until CAS(top, old, new)
8:     return old
```

Figure 2.6 shows one of many problem scenarios. In (a), our stack contains the elements A and C. Suppose that thread 1 begins to execute pop(&top), and has completed line 6, but has yet to reach line 7. If thread 2 now executes a (complete) pop(&top) operation, followed by push(&top, &B) and then push(&top, &A), it will leave the stack as shown in (b). If thread 1 now continues, its CAS will succeed, leaving the stack in the broken state shown in (c).

The problem here is that top changed between thread 1's load and the subsequent CAS. If these two instructions were replaced with LL and SC, the latter would fail—as indeed it should—causing thread 1 to try again.

On machines with CAS, programmers must consider whether the ABA problem can arise in the algorithm at hand and, if so, take measures to avoid it. The simplest and most common technique is to devote part of each to-be-CASed word to a sequence number that is updated in pop on a successful CAS. Using this *counted pointer* technique, we can convert our stack code to the (now safe) version shown in Figure 2.7.[2]

The sequence number solution to the ABA problem requires that there be enough bits available for the number that wrap-around cannot occur in any reasonable program execution. Some machines (e.g., the x86, or the SPARC when running in 32-bit mode) provide a double-width CAS that is ideal for this purpose. If the maximum word width is required for "real" data, however, another approach may be required.

[2]While Treiber's technical report [Treiber, 1986] is the standard reference for the nonblocking stack algorithm, the ABA problem is mentioned as early as the 1975 edition of the System 370 manual [IBM, 1975, p. 125], and a version of the stack appears in the 1983 edition [IBM, 1983, App. A]. Treiber's personal contribution (not shown in Figure 2.7) was to observe that counted pointers are required only in the pop operation; push can safely perform a single-width CAS on the pointer alone [Michael, 2013].

```
class stack
    ⟨node*, int⟩ top

void stack.push(node* n):
    repeat
        ⟨o, c⟩ := top
        n→next := o
    until CAS(&top, ⟨o, c⟩, ⟨n, c⟩)

node* stack.pop():
    repeat
        ⟨o, c⟩ := top
        if o = null return null
        n := o→next
    until CAS(&top, ⟨o, c⟩, ⟨n, c+1⟩)
    return o
```

Figure 2.7: The lock-free "Treiber stack," with a counted top-of-stack pointer to solve the ABA problem. It suffices to modify the count in pop only; if CAS is available in multiple widths, it may be applied to only the pointer in push.

In many programs, the programmer can reason that a given pointer will reappear in a given data structure only as a result of memory deallocation and reallocation. Note that this is *not* the case in the Treiber stack as presented here. It would be the case if we re-wrote the code to pass push a value, and had the method allocate a new node to hold it. Symmetrically, pop would deallocate the node and return the value it contained. In a garbage-collected language, deallocation will not occur so long as any thread retains a reference, so all is well. In a language with manual storage management, *hazard pointers* [Herlihy et al., 2005, Michael, 2004b] or *read-copy-update* [McKenney et al., 2001] (Section 6.3) can be used to delay deallocation until all concurrent uses of a datum have completed. In the general case (where a pointer can recur without its memory having been recycled), safe CASing may require an extra level of pointer indirection [Jayanti and Petrovic, 2003, Michael, 2004c].

2.3.2 OTHER SYNCHRONIZATION HARDWARE

Several historic machines have provided special locking instructions. The QOLB (queue on lock bit) instruction, originally designed for the Wisconsin Multicube [Goodman et al., 1989], and later adopted for the IEEE Scalable Coherent Interface (SCI) standard [Aboulenein et al., 1994], leverages a coherence protocol that maintains a linked list of copies of a given cache line. When multiple processors attempt to lock the same line at the same time, the hardware arranges to grant the requests in linked-list order. The Kendall Square KSR-1 machine [KSR, 1992] provided a similar mechanism based not on an explicit linked list, but on the implicit ordering of nodes in a ring-based network topology. As we shall see in Chapter 4, similar strategies can be emulated in software. The principal argument for the hardware approach is the ability to avoid a costly cache miss when passing the lock (and perhaps its associated data) from one processor to the next [Woest and Goodman, 1991].

The x86 allows any memory-update instruction (e.g., add or increment) to be prefixed with a special LOCK code, rendering it atomic. The benefit to the programmer is limited, however, by the fact that most instructions do not return the previous value from the modified location:

two threads executing concurrent LOCKed increments, for example, could be assured that both operations would occur, but could not tell which happened first.

Several supercomputer-class machines have provided special network hardware (generally accessed via memory-mapped I/O) for near-constant-time barrier and "Eureka" operations. These amount to cross-machine AND (all processors are ready) and OR (some processor is ready) computations. We will mention them again in Sections 5.2 and 5.3.3.

In 1991, Stone et al. proposed a multi-word variant of LL/SC, which they called "Oklahoma Update" [Stone et al., 1993] (in reference to the song "All Er Nuthin' " from the Rodgers and Hammerstein musical). Concurrently and independently, Herlihy and Moss proposed a similar mechanism for *Transactional Memory* (TM) [Herlihy and Moss, 1993]. Neither proposal was implemented at the time, but TM enjoyed a rebirth of interest about a decade later. Like queued locks, it can be implemented in software using CAS or LL/SC, but hardware implementations enjoy a substantial performance advantage [Harris et al., 2010]. As of early 2013, hardware TM has been implemented on the Azul Systems Vega 2 and 3 [Click Jr., 2009]; the experimental Sun/Oracle Rock processor [Dice et al., 2009]; the IBM Blue Gene/Q [Wang et al., 2012], zEC12 mainframe [Jacobi et al., 2012], and Power 8 processors (three independent implementations); and Intel's "Haswell" version of the x86. Additional implementations are likely to be forthcoming. We will discuss TM in Chapter 9.

Type-preserving Allocation

Both general-purpose garbage collection and hazard pointers can be used to avoid the ABA problem in applications where it might arise due to memory reallocation. Counted pointers can be used to avoid the problem in applications (like the Treiber stack) where it might arise for reasons other than memory reallocation. But counted pointers can also be used in the presence of memory reclamation. In this case, one must employ a *type-preserving* allocator, which ensures that a block of memory is reused only for an object of the same type and alignment. Suppose, for example, that we modify the Treiber stack, as suggested in the main body of the lecture, to pass push a value, and have the method allocate a new node to hold it. In this case, if a node were deallocated and reused by unrelated code (in, say, an array of floating-point numbers), it would be possible (if unlikely) that one of those numbers might match the bit pattern of a counted pointer from the memory's former life, leading the stack code to perform an erroneous operation. With a type-preserving allocator, space once occupied by a counted pointer would continue to hold such a pointer even when reallocated, and (absent wrap-around), a CAS would succeed only in the absence of reuse.

One simple implementation of a type-preserving allocator employs a Treiber stack as a free list: old nodes are pushed onto the stack when freed; new nodes are popped from the stack, or, if the stack is empty, obtained from the system memory manager. A more sophisticated implementation avoids unnecessary cache misses and contention on the top-of-stack pointer by employing a separate pool of free nodes for each thread or core. If the local pool is empty, a thread obtains a new "batch" of nodes from a backup central pool, or, if it is empty, the system memory manager. If the local pool grows too large (e.g., in a program that performs most enqueues in one thread and most dequeues in another), a thread moves a batch of nodes back to the central pool. The central pool is naturally implemented as a Treiber stack of batches.

CHAPTER 3

Essential Theory

Concurrent algorithms and synchronization techniques have a long and very rich history of formalization—far too much to even survey adequately here. Arguably the most accessible resource for practitioners is the text of Herlihy and Shavit [2008]. Deeper, more mathematical coverage can be found in the text of Schneider [1997]. On the broader topic of distributed computing (which as noted in the box on page 2 is viewed by theoreticians as a superset of shared-memory concurrency), interested readers may wish to consult the classic text of Lynch [1996].

For the purposes of the current text, we provide a brief introduction here to *safety*, *liveness*, the *consensus hierarchy*, and formal *memory models*. Safety and liveness were mentioned briefly in Section 1.4. The former says that bad things never happen; the latter says that good things eventually do. The consensus hierarchy explains the relative expressive power of hardware primitives like test_and_set (TAS) and compare_and_swap (CAS). Memory models explain which writes may be seen by which reads under which circumstances; they help to regularize the "out of order" memory references mentioned in Section 2.2.

3.1 SAFETY

Most concurrent data structures (objects) are adaptations of sequential data structures. Each of these, in turn, has its own *sequential semantics*, typically specified as a set of preconditions and postconditions for each of the methods that operate on the structure, together with *invariants* that all the methods must preserve. The sequential implementation of an object is considered safe if each method, called when its preconditions are true, terminates after a finite number of steps, having ensured the postconditions and preserved the invariants.

When designing a concurrent object, we typically wish to allow concurrent method calls ("operations"), each of which should appear to occur atomically. This goal in turn leads to at least three safety issues:

1. In a sequential program, an attempt to call a method whose precondition does not hold can often be considered an error: the program's single thread has complete control over the order in which methods are called, and can either reason that a given call is valid or else check the precondition first, explicitly, without worrying about changes between the check and the call (if (¬Q.empty()) e := Q.dequeue()). In a parallel program, the potential for concurrent operation in other threads generally requires either that a method be *total* (i.e., that its precondition simply be true, allowing it to run under any circumstances), or that it use condition synchronization to wait until the precondition holds. The former option

is trivial if we are willing to return an indication that the operation is not currently valid (Q.dequeue(), for example, might return a special $\perp$ value when the queue is empty). The latter option is explored in Chapter 5.

2. Because threads may wait for one another due to locking or condition synchronization, we must address the possibility of *deadlock*, in which some set of threads are permanently waiting for each other. We consider lock-based deadlock in Section 3.1.1. Deadlocks due to condition synchronization are a matter of application-level semantics, and must be addressed on a program-by-program basis.

3. The notion of atomicity requires clarification. If operations do not actually execute one at a time in mutual exclusion, we must somehow specify the order(s) in which they are permitted to *appear* to execute. We consider several popular notions of ordering, and the differences among them, in Section 3.1.2.

3.1.1 DEADLOCK FREEDOM

As noted in Section 1.4, deadlock freedom is a safety property: it requires that there be no reachable state of the system in which some set of threads are all "waiting for one another." As originally observed by Coffman et al. [1971], deadlock requires four simultaneous conditions:

exclusive use – threads require access to some sort of non-sharable "resources"

hold and wait – threads wait for unavailable resources while continuing to hold resources they have already acquired

irrevocability – resources cannot be forcibly taken from threads that hold them

circularity – there exists a circular chain of threads in which each is holding a resource needed by the next

In shared-memory parallel programs, "non-sharable resources" often correspond to portions of a data structure, with access protected by mutual exclusion ("mutex") locks. Given that exclusive use is fundamental, deadlock can then be addressed by breaking any one of the remaining three conditions. For example:

1. We can break the hold-and-wait condition by requiring a thread that wishes to perform a given operation to request all of its locks at once. This approach is impractical in modular software, or in situations where the identities of some of the locks depend on conditions that cannot be evaluated without holding other locks (suppose, for example, that we wish to move an element atomically from set A to set $f(v)$, where v is the value of the element drawn from set A).

2. We can break the irrevocability condition by requiring a thread to release any locks it already holds when it tries to acquire a lock that is held by another thread. This approach is

commonly employed (automatically) in transactional memory systems, which are able to "back a thread out" and retry an operation (transaction) that encounters a locking conflict. It can also be used (more manually) in any system capable of dynamic *deadlock detection* (see, for example, the recent work of Koskinen and Herlihy [2008]). Retrying is complicated by the possibility that an operation may already have generated externally-visible side effects, which must be "rolled back" without compromising global invariants. We will consider roll-back further in Chapter 9.

3. We can break the circularity condition by imposing a static order on locks, and requiring that every operation acquire its locks according to that static order. This approach is slightly less onerous than requiring a thread to request all its locks at once, but still far from general. It does not, for example, provide an acceptable solution to the "move from A to $f(v)$" example in strategy 1 above.

Strategy 3 is widely used in practice. It appears, for example, in every major operating system kernel. The lack of generality, however, and the burden of defining—and respecting—a static order on locks, makes strategy 2 quite appealing, particularly when it can be automated, as it typically is in transactional memory. An intermediate alternative, sometimes used for applications whose synchronization behavior is well understood, is to consider, at each individual lock request, whether there is a feasible order in which currently active operations might complete (under worst-case assumptions about the future resources they might need in order to do so), even if the current lock is granted. The best known strategy of this sort is the *Banker's algorithm* of Dijkstra [early 1960s, 1982], originally developed for the THE operating system [Dijkstra, 1968a]. Where strategies 1 and 3 may be said to *prevent* deadlock by design, the Banker's algorithm is often described as deadlock *avoidance*, and strategy 2 as deadlock *recovery*.

3.1.2 ATOMICITY

In Section 2.2 we introduced the notion of *sequential consistency*, which requires that low-level memory accesses appear to occur in some global total order—i.e., "one at a time"—with each core's accesses appearing in program order (the order specified by the core's sequential program). When considering the order of high-level operations on a concurrent object, it is tempting to ask whether sequential consistency can help. In one sense, the answer is clearly no: correct sequential code will typically not work correctly when executed (without synchronization) by multiple threads concurrently—even on a system with sequentially consistent memory. Conversely, as we shall see in Section 3.4, one can (with appropriate synchronization) build correct high-level objects on top of a system whose memory is more relaxed.

At the same time, the notion of sequential consistency suggests a way in which we might define atomicity for a concurrent object, allowing us to infer what it means for code to be properly synchronized. After all, the memory system is a complex concurrent object from the perspective of a memory architect, who must implement `load` and `store` instructions via messages across a

distributed cache-cache interconnect. Just as the designer of a sequentially consistent memory system might seek to achieve the appearance of a total order on memory accesses, consistent with per-core program order, so too might the designer of a concurrent object seek to achieve the appearance of a total order on high-level operations, consistent with the order of each thread's sequential program. In any execution that appeared to exhibit such a total order, each operation could be said to have executed atomically.

Sequential Consistency for High-Level Objects

The implementation of a concurrent object O is said to be *sequentially consistent* if, in every possible execution, the operations on O appear to occur in (have the same arguments and return values that they would have had in) some total order that is consistent with program order in each thread. Unfortunately, there is a problem with sequential consistency that limits its usefulness for high-level concurrent objects: lack of *composable orders*.

A multiprocessor memory system is, in effect, a single concurrent object, designed at one time by one architectural team. Its methods are the memory access instructions. A high-level concurrent object, by contrast, may be designed in isolation, and then used with other such objects in a single program. Suppose we have implemented object A, and have proved that in any given program, operations performed on A will appear to occur in some total order consistent with program order in each thread. Suppose we have a similar guarantee for object B. We should like to be able to guarantee that in any given program, operations on A and B will appear to occur in some *single* total order consistent with program order in each thread. That is, we should like the implementations of A and B to *compose*. Sadly, they may not.

As a simple if somewhat contrived example, consider a replicated integer object, in which threads read their local copy (without synchronization) and update all copies under protection of a lock:

```
// initially L is free and A[i] = 0 ∀ i ∈ 𝒯

void put(int v):                    int get():
    L.acquire()                         return A[self]
    for i ∈ 𝒯
        A[i] := v
    L.release()
```

(Throughout this lecture, we use $\mathcal{T}$ to represent the set of thread ids. For the sake of convenience, we assume that the set is sufficiently dense that we can use it to index arrays.)

Because of the lock, put operations are totally ordered. Further, because a get operation performs only a single (atomic) access to memory, it is easily ordered with respect to all puts—after those that have updated the relevant element of A, and before those that have not. It is straightforward to identify a total order on operations that respects these constraints and that is consistent with program order in each thread. In other words, our counter is sequentially consistent.

On the other hand, consider what happens if we have *two* counters—call them X and Y. Because get operations can occur "in the middle of" a put at the implementation level, we can imagine a scenario in which threads $T3$ and $T4$ perform gets on X and Y while both objects are being updated—and see the updates in opposite orders:

	local values of shared objects			
	$T1$	$T2$	$T3$	$T4$
	X Y	X Y	X Y	X Y
initially	0 0	0 0	0 0	0 0
$T1$ begins X.put(1)	1 0	1 0	0 0	0 0
$T2$ begins Y.put(1)	1 1	1 1	0 1	0 0
$T3$: X.get() returns 0	1 1	1 1	0 1	0 0
$T3$: Y.get() returns 1	1 1	1 1	0 1	0 0
$T1$ finishes X.put(1)	1 1	1 1	1 1	1 0
$T4$: X.get() returns 1	1 1	1 1	1 1	1 0
$T4$: Y.get() returns 0	1 1	1 1	1 1	1 0
$T2$ finishes Y.put(1)	1 1	1 1	1 1	1 0

At this point, the put to Y has happened before the put to X from $T3$'s perspective, but after the put to X from $T4$'s perspective. To solve this problem, we might require the implementation of a shared object to ensure that updates appear to other threads to happen at some single point in time.

But this is not enough. Consider a software emulation of the hardware *write buffers* described in Section 2.2.1. To perform a put on object X, thread T inserts the desired new value into a local queue and continues execution. Periodically, a helper thread drains the queue and applies the updates to the master copy of X, which resides in some global location. To perform a get, T inspects the local queue (synchronizing with the helper as necessary) and returns any pending update; otherwise it returns the global value of X. From the point of view of every thread other than T, the update occurs when it is applied to the global value of X. From T's perspective, however, it happens early, and, in a system with more than one object, we can easily obtain the "bow tie" causality loop of Figure 2.3. This scenario suggests that we require updates to appear to other threads at the same time they appear to the updater—or at least before the updater continues execution.

Linearizability

To address the problem of composability, Herlihy and Wing introduced the notion of *linearizability* [1990]. For more than 20 years it has served as the standard ordering criterion for high-level concurrent objects. The implementation of object O is said to be linearizable if, in every possible execution, the operations on O appear to occur in some total order that is consistent not only

with program order in each thread but also with any ordering that threads are able to observe by other means.

More specifically, linearizability requires that each operation appear to occur instantaneously at some point in time between its call and return. The "instantaneously" part of this requirement precludes the shared counter scenario above, in which $T3$ and $T4$ have different views of partial updates. The "between its call and return" part of the requirement precludes the software write buffer scenario, in which a put by thread T may not be visible to other threads until after it has returned.

For the sake of precision, it should be noted that there is no absolute notion of objective time in a parallel system, any more than there is in Einsteinian physics. (For more on the notion of time in parallel systems, see the classic paper by Lamport [1978].) What really matters is observable orderings. When we say that an event must occur at a single instant in time, what we mean is that it must be impossible for thread A to observe that an event has occurred, for A to subsequently communicate with thread B (e.g., by writing a variable that B reads), and then for B to observe that the event has not yet occurred.

To help us reason about the linearizability of a concurrent object, we typically identify a *linearization point* within each method at which a call to that method can be said to have occurred. If we choose these points properly, then whenever the linearization point of operation A precedes the linearization point of operation B, we will know that operation A, as a whole, linearizes before operation B.

In the trivial case in which every method is bracketed by the acquisition and release of a common object lock, the linearization point can be anywhere inside the method—we might arbitrarily place it at the lock release. In an algorithm based on fine-grain locks, the linearization point might correspond to the release of some particular one of the locks.

In nonblocking algorithms, it is common to associate linearization with a specific instruction (a load, store, or other atomic primitive) and then argue that any implementation-level memory updates that are visible before the linearization point will be recognized by other threads as merely preparation, and any that can be seen to occur after it will be recognized as merely cleanup. In the nonblocking stack of Figure 2.7, a successful push or pop can be said to linearize at its final CAS instruction; an unsuccessful pop (one that returns null) can be said to linearize at the load of top.

In a complex method, we may need to identify multiple possible linearization points, to accommodate branching control flow. In other cases, the outcome of tests at run time may allow us to argue that a method linearized at some point *earlier* in its execution (an example of this sort can be found in Section 8.3). There are even algorithms in which the linearization point of a method is determined by behavior in some *other* thread. All that really matters is that there be a total order on the linearization points, and that the behavior of operations, when considered in that order, be consistent with the object's sequential semantics.

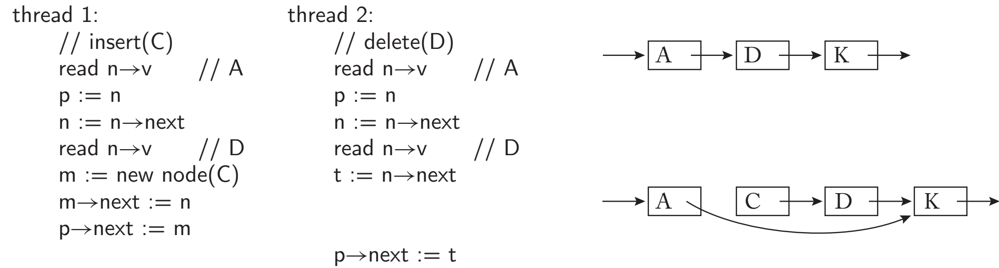


Figure 3.1: Dynamic trace of improperly synchronized list updates. This execution can lose node C even on a sequentially consistent machine.

Given linearizable implementations of objects A and B, one can prove that in every possible program execution, the operations on A and B will appear to occur in some *single* total order that is consistent both with program order in each thread and with any other ordering that threads are able to observe. In other words, linearizable implementations of concurrent objects are composable. Linearizability is therefore sometimes said to be a *local* property [Herlihy and Wing, 1990, Weihl, 1989]: the linearizability of a system as a whole depends only on the (local) linearizability of its parts.

Hand-over-hand Locking (Lock Coupling). As an example of linearizability achieved through fine-grain locking, consider the task of parallelizing a set abstraction implemented as a sorted, singly-linked list with insert, remove, and lookup operations. Absent synchronization, it is easy to see how the list could become corrupted. In Figure 3.1, the code at left shows a possible sequence of statements executed by thread 1 in the process of inserting a new node containing the value C, and a concurrent sequence of statements executed by thread 2 in the process of deleting the node containing the value D. If interleaved as shown (with thread 1 performing its last statement between thread 2's last two statements), these two sequences will transform the list at the upper right into the non-list at the lower right, in which the node containing C has been lost.

Clearly a global lock—forcing either thread 1 or thread 2 to complete before the other starts—would linearize the updates and avoid the loss of C. It can be shown, however, that linearizability can also be maintained with a fine-grain locking protocol in which each thread holds at most two locks at a time, on adjacent nodes in the list [Bayer and Schkolnick, 1977]. By retaining the right-hand lock while releasing the left-hand and then acquiring the right-hand's successor, a thread ensures that it is never overtaken by another thread during its traversal of the list. In Figure 3.1, thread 1 would hold locks on the nodes containing A and D until done inserting the node containing C. Thread 2 would need these same two locks before reading the content of the node containing A. While threads 1 and 2 cannot make their updates simultaneously, one can "chase the other down the list" to the point where the updates are needed, achieving substantially higher

concurrency than is possible with a global lock. Similar "hand-over-hand" locking techniques are commonly used in concurrent trees and other pointer-based data structures.

Serializability

Recall that the purpose of an ordering criterion is to clarify the meaning of atomicity. By requiring an operation to complete at a single point in time, and to be visible to all other threads before it returns to its caller, linearizability guarantees that the order of operations on any given concurrent object will be consistent with all other observable orderings in an execution, including those of other concurrent objects.

The flip side of this guarantee is that the linearizability of individual operations does not necessarily imply linearizability for operations that manipulate more than one object, but are still intended to execute as a single atomic unit.

Consider a banking system in which thread 1 transfers $100 from account A to account B, while thread 2 adds the amounts in the two accounts:

```
// initially A.balance() = B.balance() = 500
thread 1:                                thread 2:
    A.withdraw(100)
                                             sum := A.balance()      // 400
                                             sum +:= B.balance()     // 900
    B.deposit(100)
```

If we think of A and B as separate objects, then the execution can linearize as suggested by vertical position on the page, but thread 2 will see a cross-account total that is $100 "too low." If we wish to treat the code in each thread as a single atomic unit, we must disallow this execution—something that neither A nor B can do on its own. We need, in short, to be able to *combine* smaller atomic operations into larger ones—not just perform the smaller ones in a mutually consistent order. Where linearizability ensures that the orders of separate objects will compose "for free," multi-object atomic operations will generally require some sort of global or distributed control.

Multi-object atomic operations are the hallmark of database systems, which refer to them as *transactions*. Transactional memory (the subject of Chapter 9) adapts transactions to shared-memory parallel computing, allowing the programmer to request that a multi-object operation like thread 1's transfer or thread 2's sum should execute atomically.

The simplest ordering criterion for transactions—both database and memory—is known as *serializability*. Transactions are said to serialize if they have the same effect they would have had if executed one at a time in some total order. For transactional memory (and sometimes for databases as well), we can extend the model to allow a thread to perform a series of transactions, and require that the global order be consistent with program order in each thread.

It turns out to be NP-hard to determine whether a given set of transactions (with the given inputs and outputs) is serializable [Papadimitriou, 1979]. Fortunately, we seldom need to make such a determination in practice. Generally all we really want is to ensure that the current

execution will be serializable—something we can achieve with conservative (sufficient but not necessary) measures. A global lock is a trivial solution, but admits no concurrency. Databases and most TM systems employ more elaborate fine-grain locking. A few TM systems employ nonblocking techniques.

If we regard the objects to be accessed by a transaction as "resources" and revisit the conditions for deadlock outlined at the beginning of Section 3.1.1, we quickly realize that a transaction may, in the general case, need to access some resources before it knows which others it will need. Any implementation of serializability based on fine-grain locks will thus entail not only "exclusive use," but also both "hold and wait" and "circularity." To address the possibility of deadlock, a database or lock-based TM system must be prepared to break the "irrevocability" condition by releasing locks, rolling back, and retrying conflicting transactions.

Like branch prediction or CAS-based fetch_and_Φ, this strategy of proceeding "in the hope" that things will work out (and recovering when they don't) is an example of *speculation*. So-called *lazy* TM systems take this even further, allowing conflicting (non-serializable) transactions to proceed in parallel until one of them is ready to *commit*—and only then *aborting* and rolling back the others.

Two-Phase Locking. As an example of fine-grain locking for serializability, consider a simple scenario in which transactions 1 and 2 read and update symmetric variables:

```
// initially x = y = 0
transaction 1:                    transaction 2:
    t1 := x                           t2 := y
    y++                               x++
```

Left unsynchronized, this code could result in t1 = t2 = 0, even on a sequentially consistent machine—something that should not be possible if the transactions are to serialize. A global lock would solve the problem, but would be far too conservative for transactions larger than the trivial ones shown here. If we associate fine-grain locks with individual variables, we still run into trouble if thread 1 releases its lock on x before acquiring the lock on y, and thread 2 releases its lock on y before acquiring the lock on x.

It turns out [Eswaran et al., 1976] that serializability can always be guaranteed if threads acquire all their locks (in an "expansion phase") before releasing any of them (in a "contraction phase"). As we have observed, this *two-phase locking* convention admits the possibility of deadlock: in our example, transaction 1 might lock x and transaction 2 lock y before either attempts to acquire the other. To detect the problem and trigger rollback, a system based on two-phase locking may construct and maintain a dependence graph at run time. Alternatively (and more conservatively), it may simply limit the time it is willing to wait for locks, and assume the worst when this timeout is exceeded.

Strict Serializability

The astute reader may have noticed the strong similarity between the definitions of sequential consistency (for high-level objects) and serializability (with the extension that allows a single thread to perform a series of transactions). The difference is simply that transactions need to be able to access a dynamically chosen set of objects, while sequential consistency is limited to a predefined set of single-object operations.

The similarity between sequential consistency and serializability leads to a common weakness: the lack of required consistency with other orders that may be observed by a thread. It was by requiring such "real-time" ordering that we obtained composable orders for single-object operations in the definition of linearizability. Real-time ordering is also important for its own sake in many applications. Without it we might, for example, make a large deposit to a friend's bank account, tell the person we had done so, and yet still encounter an "insufficient funds" message in response to a (subsequent!) withdrawal request. To avoid such arguably undesirable scenarios, many database systems—and most TM systems—require *strict serializability*, which is simply ordinary serializability augmented with real-time order: transactions are said to be strictly serializable if they have the same effect they would have had if executed one at a time in some total order that is consistent with program order (if any) in each thread, and with any other order the threads may be able to observe. In particular, if transaction A finishes before transaction B begins, then A must appear before B in the total order. As it turns out, two-phase locking suffices to ensure strict serializability, but certain other implementations of "plain" serializability do not.

Relationships Among the Ordering Criteria

Figure 3.1 summarizes the relationships among sequential consistency (for high-level objects), linearizability, serializability, and strict serializability. A system that correctly implements any of these four criteria will provide the appearance of a total order on operations, consistent with per-thread program order. Linearizability and strict serializability add consistency with "real-time" order. Serializability and strict serializability add the ability to define multi-object atomic operations. Of the four criteria, only linearizability is local: it guarantees that operations on separate objects always occur in a mutually consistent order, and it declines, as it were, to address multi-object operations.

To avoid confusion, it should be noted that we have been using the term "composability" to mean that we can merge (compose) the orders of operations on separate objects into a single mutually consistent order. In the database and TM communities, "composability" means that we can combine (compose) individual atomic operations into larger, still atomic (i.e., serializable) operations. We will return to this second notion of composability in Chapter 9. It is straightforward to provide in a system based on speculation; it is invariably supported by databases and transactional memory. It cannot be supported, in the general case, by conservative locking strategies. Somewhat ironically, linearizability might be said to facilitate composable orders by disallowing composable operations.

Table 3.1: Properties of standard ordering criteria

SC = sequential consistency; L = linearizability;
S = serializability; SS = strict serializability.

	SC	L	S	SS
Equivalent to a sequential order	+	+	+	+
Respects program order in each thread	+	+	+	+
Consistent with other ordering ("real time")	−	+	−	+
Can touch multiple objects atomically	−	−	+	+
Local: reasoning based on individual objects only	−	+	−	−

3.2 LIVENESS

Safety properties—the subject of the previous section—ensure that bad things never happen: threads are never deadlocked; atomicity is never violated; invariants are never broken. To say that code is correct, however, we generally want more: we want to ensure forward progress. Just as we generally want to know that a sequential program will produce a correct answer eventually (not just fail to produce an incorrect answer), we generally want to know that invocations of concurrent operations will complete their work and return.

An object method is said to be *blocking* (in the theoretical sense described in the box on page 7) if there is some reachable state of the system in which a thread that has called the method will be unable to return until some other thread takes action. Lock-based algorithms are inherently blocking: a thread that holds a lock precludes progress on the part of any other thread that needs the same lock. Liveness proofs for lock-based algorithms require not only that the code be deadlock-free, but also that critical sections be free of infinite loops, and that all threads continue to execute.

A method is said to be *nonblocking* if there is no reachable state of the system in which an invocation of the method will be unable to complete its execution and return. Nonblocking algorithms have the desirable property that inopportune preemption (e.g., of a lock holder) never precludes forward progress in other threads. In some environments (e.g., a system with high fault-tolerance requirements), nonblocking algorithms may also allow the system to survive when a thread crashes or is prematurely killed. We consider several variants of nonblocking progress in Section 3.2.1.

In both blocking and nonblocking algorithms, we may also care about *fairness*—the relative rates of progress of different threads. We consider this topic briefly in Section 3.2.2.

3.2.1 NONBLOCKING PROGRESS

Given the difficulty of guaranteeing any particular rate of execution (in the presence of timesharing, cache misses, page faults, and other sources of variability), we generally speak of progress in terms of abstract program *steps* rather than absolute time.

A method is said to be *wait free* (the strongest variant of nonblocking progress) if it is guaranteed to complete in some bounded number of its own program steps. (This bound need not be statically known.) A method M is said to be *lock free* (a somewhat weaker variant) if *some* thread is guaranteed to make progress (complete an operation on the same object) in some bounded number of M's program steps. M is said to be *obstruction free* (the weakest variant of nonblocking progress) if it is guaranteed to complete in some bounded number of program steps if no other thread executes any steps during that same interval.

Wait freedom is sometimes referred to as *starvation freedom*: a given thread is never prevented from making progress. Lock freedom is sometimes referred to as *livelock freedom*: an individual thread may starve, but the system as a whole is never prevented from making forward progress (equivalently: no set of threads can actively prevent each other from making progress indefinitely). Obstruction-free algorithms can suffer not only from starvation but also from livelock; if all threads but one "hold still" long enough, however, the one running thread is guaranteed to make progress.

Many practical algorithms are lock free or obstruction free. Treiber's stack, for example (Section 2.3.1), is lock-free, as is the widely used queue of Michael and Scott (Section 8.2). Obstruction freedom was first described in the context of Herlihy et al.'s double-ended queue [2003a] (Section 8.6.2). It is also provided by several TM systems (among them the DSTM of Herlihy et al. [2003b], the ASTM of Marathe et al. [2005], and the work of Marathe and Moir [2008]). Moir and Shavit [2005] provide an excellent survey of concurrent data structures, including coverage of nonblocking progress. Sundell and Tsigas [2008a] describe a library of nonblocking data structures.

Wait-free algorithms are significantly less common. Herlihy [1991] demonstrated that any sequential data structure can be transformed, automatically, into a wait-free concurrent version, but the construction is highly inefficient. Recent work by Kogan and Petrank [2012] (building on a series of intermediate results) has shown how to reduce the time overhead dramatically, though space overhead remains proportional to the maximum number of threads in the system.

Most wait-free algorithms—and many lock-free algorithms—employ some variant of *helping*, in which a thread that has begun but not completed its operation may be assisted by other threads, which need to "get it out of the way" so they can perform their own operations without an unbounded wait. Other lock-free algorithms—and even a few wait-free algorithms—are able to make do without helping. As a simple example, consider the following implementation of a wait-free increment-only counter:

```
initially C[i] = 0 ∀ i ∈ 𝒯

void inc():
    C[self]++

int val():
    rtn := 0
    for i in [1..N]
        rtn +:= C[i]
    return rtn
```

Here the aggregate counter value is taken to be the sum of a set of per-thread values. Because each per-thread value is monotonically increasing (and always by exactly one), so is the aggregate sum. Given this observation, one can prove that the value returned by the `val` method will have been correct at some point between its call and its return: it will be bounded above by the number of `inc` operations that were called before `val` was called, and below by the number that returned before it was called. In other words, `val` is linearizable, though its linearization point cannot in general be statically determined. Because both `inc` and `val` comprise a bounded (in this case, statically bounded) number of program steps, the methods are wait free.

3.2.2 FAIRNESS

Obstruction freedom and lock freedom clearly admit behavior that defies any notion of fairness: both allow an individual thread to take an unbounded number of steps without completing an operation. Even wait freedom allows an operation to execute an arbitrary number of steps (helping or deferring to peers) before completing, so long as the number is bounded in any given situation.

We shall often want stronger guarantees. In a wait-free algorithm, we might hope for a static bound, across all invocations, on the number of steps required to complete an operation. In a blocking algorithm, we might hope for a bound on the number of competing operations that may complete before a given thread makes progress. If threads repeatedly invoke a certain set of operations, we might even wish to bound the ratio of their "success" rates. These are only a few of the possible ways in which "fairness" might be defined. Without dwelling on particular definitions, we will consider algorithms in subsequent chapters whose behavior ranges from potentially very highly skewed (e.g., `test_and_set` locks that avoid starvation only when there are periodic quiescent intervals, when the lock is free and no thread wants it), to strictly first-come, first-served (e.g., locks in which a thread employs a wait-free protocol to join a FIFO queue). We will also consider intermediate options, such as locks that deliberately balance locality (for performance) against uniformity of service to threads.

In any practical system, forward progress relies on the assumption that any continually unblocked thread will eventually execute another program step. Without such minimal fairness within the implementation, a system could be "correct" without doing anything at all! Significantly, even this minimal fairness depends on scheduling decisions at multiple system levels—in the hardware, the operating system, and the language runtime—all of which ensure that runnable threads continue to run.

When threads may block for mutual exclusion or condition synchronization, we shall in most cases want to insist that the system display what is known as *weak fairness*. This property guarantees that any thread waiting for a condition that is continuously true (or a lock that is continuously available) eventually executes another program step. Without such a guarantee, program behavior may be highly unappealing. Imagine a web server, for example, that never accepts requests from a certain client connection if requests are available from any other client.

In the following program fragment, weak fairness precludes an execution in which thread 1 spins forever: thread 2 must eventually notice that f is false, complete its wait, and set f to true, after which thread 1 must notice the change to f and complete:

```
initially f = false

thread 1:                        thread 2:
    await f                          await ¬f
                                     f := true
```

Here we have used the notation await (*condition*) as shorthand for

```
while ¬condition
    // spin
fence(R||RW)
```

Many more stringent definitions of fairness are possible. In particular, *strong fairness* requires that any thread waiting for a condition that is true infinitely often (or a lock that is available infinitely often) eventually executes another program step. In the following program fragment, for example, weak fairness admits an execution in which thread 1 spins forever, but strong fairness requires thread 2 to notice one of the "windows" in which g is true, complete its wait, and set f to true, after which thread 1 must notice the change and complete:

```
initially f = g = false

thread 1:                        thread 2:
    while ¬f                         await (g)
        g := true                    f := true
        g := false
```

Strong fairness is difficult to truly achieve: it may, for example, require a scheduler to re-check every awaited condition whenever one of its constituent variables is changed, to make sure that any thread at risk of starving is given a chance to run. Any deterministic strategy that considers only a subset of the waiting threads on each state change risks the possibility of deterministically ignoring some unfortunate thread every time it is able to run.

Fortunately, statistical "guarantees" typically suffice in practice. By considering a randomly chosen thread—instead of all threads—when a scheduling decision is required, we can drive the probability of starvation arbitrarily low. A truly random choice is difficult, of course, but various pseudorandom approaches appear to work quite well. At the hardware level, interconnects and coherence protocols are designed to make it unlikely that a "race" between two cores (e.g., when performing near-simultaneous CAS instructions on a previously uncached location) will always be resolved the same way. Within the operating system, runtime, or language implementation, one can "randomize" the interval between checks of a condition using a pseudorandom number generator or even the natural "jitter" in execution time of nontrivial instruction sequences on complex modern cores.

Weak and strong fairness address worst-case behavior, and allow executions that still seem grossly unfair from an intuitive perspective (e.g., executions in which one thread succeeds a million times more often than another). Statistical "randomization," by contrast, may achieve intuitively very fair behavior without absolutely precluding worst-case starvation.

Much of the theoretical groundwork for fairness was laid by Nissim Francez [1986]. Proofs of fairness are typically based on *temporal logic*, which provides operators for concepts like "always" and "eventually." A brief introduction to these topics can be found in the text of Ben-Ari [2006, Chap. 4]; much more extensive coverage can be found in Schneider's comprehensive work on the theory of concurrency [1997].

3.3 THE CONSENSUS HIERARCHY

In Section 2.3 we noted that CAS and LL/SC are *universal* atomic primitives—capable of implementing arbitrary single-word fetch_and_Φ operations. We suggested—implicitly, at least—that they are fundamentally more powerful than simpler primitives like TAS, Swap, FAI, and FAA. Herlihy formalized this notion of relative power in his work on wait-free synchronization [1991], previously mentioned in Section 3.2.1. The formalization is based on the classic *consensus problem*.

Originally formalized by Fischer, Lynch, and Paterson [1985] in a distributed setting, the consensus problem involves a set of potentially unreliable threads, each of which "proposes" a value. The goal is for the reliable threads to agree on one of the proposed values—a task the authors proved to be impossible with asynchronous messages. Herlihy adapted the problem to the shared-memory setting, where powerful atomic primitives can circumvent impossibility. Specifically, Herlihy suggested that such primitives (or, more precisely, the objects on which those primitives operate) be classified according the number of threads for which they can achieve *wait-free* consensus.

It is easy to see that an object with a TAS method can achieve wait-free consensus for two threads:

```
// initially L = 0; proposal[0] and proposal[1] are immaterial
agree(i):
    proposal[self].store(i)
    if TAS(L) return i
    else return proposal[1−self].load()
```

Herlihy was able to show that this is the best one can do: TAS objects (even an arbitrary number of them) cannot achieve wait-free consensus for more than two threads. Moreover ordinary loads and stores cannot achieve wait-free consensus at all—even for only two threads. An object supporting CAS, on the other hand (or equivalently LL/SC), can achieve wait-free consensus for an arbitrary number of threads:

```
// initially v = ⊥
agree(i):
    if CAS(&v, ⊥, i) return i
    else return v
```

One can, in fact, define an infinite hierarchy of atomic objects, where those appearing at level k can achieve wait-free consensus for k threads but no more. Objects supporting CAS or LL/SC are said to have *consensus number* ∞. Objects with other common primitives—including TAS, swap, FAI, and FAA—have consensus number 2. One can define atomic objects at intermediate levels of the hierarchy, but these are not typically encountered on real hardware.

3.4 MEMORY MODELS

As described in Section 2.2, most modern multicore systems are *coherent* but not *sequentially consistent*: changes to a given variable are serialized, and eventually propagate to all cores, but accesses to different locations may appear to occur in different orders from the perspective of different threads—even to the point of introducing apparent causality loops. For programmers to reason about such a system, we need a *memory model*—a formal characterization of its behavior. Such a model can be provided at the hardware level—SPARC TSO is one example—but when programmers write code in higher-level languages, they need a language-level model. Just as a hardware-level memory model helps the compiler writer or library builder determine where to employ special ordering instructions, a language-level model helps the application programmer determine where to employ synchronization operations or atomic variable accesses.

There is an extensive literature on language-level memory models. Good starting points include the tutorial of Adve and Gharachorloo [1996], the lecture of Sorin et al. [2011], and the articles introducing the models of Java [Manson et al., 2005] and C++ [Boehm and Adve, 2008]. Details vary considerably from one model to another, but most now share a similar framework.

Consensus and Mutual Exclusion

A solution to the consensus problem clearly suffices for "one-shot" mutual exclusion (a.k.a. *leader election*): each thread proposes its own id, and the agreed-upon value indicates which thread is able to enter the critical section. Consensus is not *necessary* however: the winning thread needs to know that it can enter the critical section, but other threads only need to know that they have lost—they don't need to know who won. TAS thus suffices to build a wait-free *try lock* (one whose acquire method returns immediately with a success or failure result) for an arbitrary number of threads.

It is tempting to suggest that one might solve the consensus problem using mutual exclusion, by having the winner of the competition for the lock write its id into a location visible to the losers. This approach, however, cannot be wait free: it fails to bound the number of steps required by a losing thread. If the winner acquires the lock and then pauses—or dies—before writing down its id, the losers may execute their spin loops an arbitrary number of times. This is the beauty of CAS or LL/SC: it allows a thread to win the competition *and* write down its value in a single indivisible step.

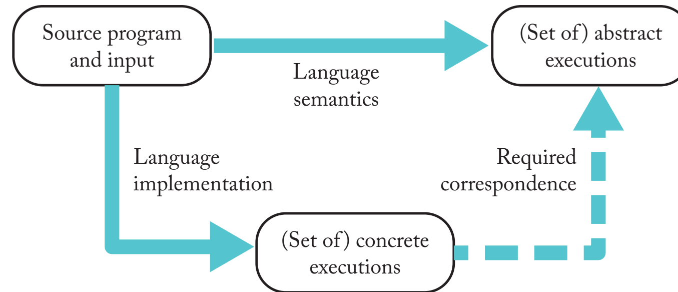


Figure 3.2: Program executions, semantics, and implementations. A valid implementation must produce only those concrete executions whose output agrees with that of some abstract execution allowed by language semantics for the given program and input.

3.4.1 FORMAL FRAMEWORK

Informally, a memory model specifies which values (i.e., which writes) may be seen by any given read. At the language level, more precise definitions depend on the notion of an *abstract program execution*.

Programming language semantics are typically defined in terms of execution on some abstract machine, with language-appropriate built-in types, control-flow constructs, etc. For a given source program and input, an abstract execution is a set of sequences, one per thread, of reads, writes, and other atomic program steps, each of which inspects and/or changes the state of the abstract machine (i.e., memory). For any given program and input, certain executions of the abstract machine are said to be *valid* according to the language's semantics. Language semantics can be seen, constructively, as a mapping from programs and inputs to (possibly infinite) sets of valid executions. The semantics can also be seen, nonconstructively, as a predicate: given a program, its input, and an abstract execution, is the execution valid?

For the sake of convenience, the remainder of this section adopts the nonconstructive point of view, at least with respect to the memory model. In practice, it is the language implementor who needs to construct an execution—a *concrete* execution—for a given program and input. A language implementation is *safe* if for every well formed source program and input, every concrete execution it produces corresponds to (produces the same output as) some valid abstract execution of that program on that input (Figure 3.2). The implementation is *live* if produces at least one such concrete execution whenever the set of abstract executions is nonempty. To show that a language implementation is safe (a task beyond the scope of this lecture), one can define a mapping from concrete to abstract executions, and then demonstrate that all the resulting abstract executions will be valid.

At the programming language level, a memory model is the portion of language semantics that determines whether the values read from variables in an abstract execution are valid, given

the values written by other program steps. In a single-threaded program, the memory model is trivial: there is a total order on program steps in any given execution, and the value read in a given step is valid if and only if it matches the value written by the most recent prior write to the same variable—or the initial value if there is no such write.

In a multithreaded program, the memory model is substantially more complex, because a read in one thread may see a value written by a write in a different thread. An execution is said to be sequentially consistent (in the sense of Section 2.2) if there exists a total order on memory operations (across all threads) that explains all values read. As we have seen, of course, most hardware is not sequentially consistent. Absent language restrictions, a concrete execution is unlikely to be both fast and sequentially consistent. For the sake of efficiency, the memory model for a concurrent language therefore typically requires only a partial order—known as *happens-before*—on the steps of an abstract execution. Using this partial order, the model then defines a *writes-seen* relation that identifies, for every read, the writes whose values may be seen.

To define the happens-before order, most memory models begin by distinguishing between "ordinary" and "synchronizing" steps in the abstract execution. Ordinary steps are typically reads and writes of scalar variables. Depending on the language, synchronizing steps might be lock `acquire` and `release`, transactions, monitor entry and exit, message send and receive, or reads and writes of special `atomic` variables. Like ordinary steps, most synchronizing steps read or write values in memory (an `acquire` operation, for example, changes a lock from "free" to "held"). A sequentially consistent execution must explain these reads and writes, just as it does those effected by accesses to ordinary scalar variables.

Given the definition of ordinary and synchronizing steps, we proceed incrementally as follows:

Program order is the union of a collection of disjoint total orders, each of which captures the steps performed by one of the program's threads. Each thread's steps must be allowable under the language's sequential semantics, given the values returned by read operations.

Synchronization order is a total order, across all threads, on all synchronizing steps. This order must be consistent with program order within each thread. It must also explain the values read and written by the synchronizing steps (this will ensure, for example, that `acquire` and `release` operations on any given lock occur in alternating order). Crucially, synchronization order is not specified by the source program. An execution is valid only if there exists a synchronization order that leads, as described below, to a writes-seen relation that explains the values read by ordinary steps.

Synchronizes-with order is a subset of synchronization order induced by language semantics. In a language based on transactional memory, the subset may be trivial: all transactions are globally ordered. In a language based on locks, each `release` operation may synchronize with the next `acquire` of the same lock in synchronization order, but other synchronizing steps may be unordered.

Happens-before order is the transitive closure of program order and synchronizes-with order. It captures all the ordering the language guarantees.

To complete a memory model, these order definitions must be augmented with a writes-seen relation. To understand such relations, we first must understand the notion of a *data race*.

3.4.2 DATA RACES

Language semantics specify classes of ordinary program steps that *conflict* with one another. A write, for example, is invariably defined to conflict with either a read or a write to the same variable. A program is said to have a data race if, for some input, it has a sequentially consistent execution in which two conflicting ordinary steps, performed by different threads, are adjacent in the total order. Data races are problematic because we don't normally expect an implementation to force ordinary steps in different threads to occur in a particular order. If an implementation yields (a concrete execution corresponding to) an abstract execution in which the conflicting steps occur in one order, we need to allow it to yield another abstract execution (of the same program on the same input) in which all prior steps are the same but the conflicting steps are reversed. It is easy to construct examples (e.g., as suggested in Figure 2.3) in which the remainder of this second execution cannot be sequentially consistent.

Given the definitions in Section 3.4.1, we can also say that an abstract execution has a data race if it contains a pair of conflicting steps that are not ordered by happens-before. A program then has a data race if, for some input, it has an execution containing a data race. This definition turns out to be equivalent to the one based on sequentially consistent executions. The key to the equivalence is the observation that arcs in the synchronizes-with order, which contribute to happens-before, correspond to ordering constraints in sequentially consistent executions. The same language rules that induce a synchronizes-with arc from, say, the release of a lock to the

---

Synchronization Races

The definition of a data race is designed to capture cases in which program behavior may depend on the order in which two ordinary accesses occur, and this order is not constrained by synchronization. In a similar fashion, we may wish to consider cases in which program behavior depends on the outcome of synchronization operations.

For each form of synchronization operation, we can define a notion of conflict. Acquire operations on the same lock, for example, conflict with one another, while an acquire and a release do not—nor do operations on different locks. A program is said to have a *synchronization race* if it has two sequentially consistent executions with a common prefix, and the first steps that differ are conflicting synchronization operations. Together, data races and synchronization races constitute the class of *general races* [Netzer and Miller, 1992].

Because we assume the existence of a total order on synchronizing steps, synchronization races never compromise sequential consistency. Rather, they provide the means of controlling and exploiting nondeterminism in parallel programs. In any case where we wish to allow conflicting high-level operations to occur in arbitrary order, we design a synchronization race into the program to mediate the conflict.

---

subsequent acquire also force the release to appear before the acquire in the total order of any sequentially consistent execution.

In an execution without any data races, the writes-seen relation is straightforward: the lack of unordered conflicting accesses implies that all reads and writes of a given location are ordered by happens-before. Each read can then return the value written by the (unique) most recent prior write of the same location in happens-before order—or the initial value if there is no such write. More formally, one can prove that all executions of a data-race-free program are sequentially consistent: any total order consistent with happens-before will explain the program's reads. Moreover, since our (first) definition of a data race was based only on sequentially consistent executions, we can provide the programmer with a set of rules that, if followed, will always lead to sequentially consistent executions, with no need to reason about possible relaxed behavior of the underlying hardware. Such a set of rules is said to constitute a *programmer-centric* memory model [Adve and Hill, 1990].

In effect, a programmer-centric model is a contract between the programmer and the implementation: if the programmer follows the rules (i.e., write data-race-free programs), the implementation will provide the illusion of sequential consistency. Moreover, given the absence of races, any region of code that contains no synchronization (and that does not interact with the "outside world" via I/O or syscalls) can be thought of as atomic: it cannot—by construction—interact with other threads.

But what about programs that *do* have data races? Some researchers have argued that such programs are simply buggy, and should have undefined behavior. This is the approach adopted by C++ [Boehm and Adve, 2008] and, subsequently, C. It rules out certain categories of programs (e.g., chaotic relaxation [Chazan and Miranker, 1969]), but the language designers had little in the way of alternatives: in the absence of type safety it is nearly impossible to limit the potential impact of a data race. The resulting model is quite simple (at least in the absence of variables that have been declared `atomic`): if a C or C++ program has a data race on a given input, its behavior is undefined; otherwise, it follows one of its sequentially consistent executions.

Unfortunately, in a language like Java, even buggy programs need to have well defined behavior, to safeguard the integrity of the virtual machine (which may be embedded in some larger, untrusting system). The obvious approach is to say that a read may see the value written by the most recent write on any backward path through the happens-before graph, or by any *incomparable* write (one that is unordered with respect to the read). Unfortunately, as described by Manson et al. [2005], this approach is overly restrictive: it precludes the use of several important compiler optimizations. The actual Java model defines a notion of "incremental justification" that may allow a read to see a value that might have been written by an incomparable write in some *other* hypothetical execution. The details are surprisingly subtle and complex, and as of 2012 it is still unclear whether the current specification is correct, or could be made so.

3.4.3 REAL-WORLD MODELS

As of this writing, Java and C/C++ are the only widely used parallel programming languages whose definitions attempt to precisely specify a memory model. Ada [Ichbiah et al., 1991] was the first language to introduce an explicitly relaxed (if informally specified) memory model. It was designed to facilitate implementation on both shared-memory and distributed hardware: variables shared between threads were required to be consistent only in the wake of explicit message passing (rendezvous). The reference implementations of several scripting languages (notably Ruby and Python) are sequentially consistent, though other implementations [JRuby, Jython] are not.

A group including representatives of Intel, Oracle, IBM, and Red Hat has proposed transactional extensions to C++ [Adl-Tabatabai et al., 2012]. In this proposal, begin_transaction and end_transaction markers contribute to the happens-before order inherited from standard C++. So-called *relaxed* transactions are permitted to contain other synchronization operations (e.g., lock acquire and release); *atomic* transactions are not. Dalessandro et al. [2010b] have proposed an alternative model in which atomic blocks are fundamental, and other synchronization mechanisms (e.g., locks) are built on top of them.

If we wish to allow programmers to create new synchronization mechanisms or nonblocking data structures (and indeed if any of the built-in synchronization mechanisms are to be written in high-level code, rather than assembler), then the memory model must define synchronizing steps that are more primitive than lock acquire and release. Java allows a variable to be labeled `volatile`, in which case reads and writes that access it are included in the global synchronization order, with each read inducing a synchronizes-with arc (and thus a happens-before arc) from the (unique) preceding write to the same location. C and C++ provide a substantially more complex facility, in which variables are labeled `atomic`, and an individual read, write, or fetch_and_Φ operation can be labeled as an acquire access, a release access, both, or neither. By default, operations on `atomic` variables are sequentially consistent: there is a global total order among them.

A crucial goal in the design of any practical memory model is to preserve, as much as possible, the freedom of compiler writers to employ code improvement techniques originally developed for sequential programs. The ordering constraints imposed by synchronization operations necessitate not only hardware-level ordered accesses or memory fences, but also software-level "compiler fences," which inhibit the sorts of code motion traditionally used for latency tolerance, redundancy elimination, etc. (Recall that in our pseudocode synchronizing instructions are intended to enforce both hardware and compiler ordering.) Much of the complexity of C/C++ `atomic` variables stems from the desire to avoid unnecessary hardware ordering and compiler fences, across a variety of hardware platforms. Within reason, programmers should attempt in C/C++ to specify the minimal ordering constraints required for correct behavior. At the same time, they should resist the temptation to "get by" with minimal ordering in the absence of a solid correctness argument. In particular, while the language allows the programmer to relax the default sequential consistency of accesses to `atomic` variables (presumably to avoid paying for write atomicity), the result can be very confusing. Recent work by Attiya et al. [2011] has also

shown that certain W‖R orderings and fetch_and_Φ operations are essential in a fundamental way: standard concurrent objects cannot be written without them.

APC: A Performance Metric of Memory Systems

Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA 60616
sun@iit.edu

Dawei Wang
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA 60616
dwang31@iit.edu

ABSTRACT
Due to the infamous "memory wall" problem and a drastic increase in the number of data intensive applications, memory rather than processor has become the leading performance bottleneck of modern computing systems. Evaluating and understanding memory system performance is increasingly becoming the core of high-end computing. Conventional memory metrics, such as miss ratio, average miss latency, average memory access time, etc., are designed to measure a given memory performance parameter, and do not reflect the overall performance of a memory system. On the other hand, widely used system measurement metrics, such as IPC and Flops are designed to measure CPU performance, and do not directly reflect memory performance. In this paper, we proposed a novel memory metric, Access Per Cycle (APC), to measure overall memory performance with consideration of the complexity of modern memory systems. A unique contribution of APC is its separation of memory evaluation from CPU evaluation; therefore, it provides a quantitative measurement of the "data-intensiveness" of an application. The concept of APC is introduced; a constructive investigation counting the number of data accesses and access cycles at differing levels of the memory hierarchy is conducted; finally some important usages of APC are presented. Simulation results show that APC is significantly more appropriate than the existing memory metrics in evaluating modern memory systems.

Keywords
Memory Performance Measurement, Memory Metric, Measurement Methodology

1. INTRODUCTION
The rapid advances of semiconductor technology have driven large increases in processor performance over the past thirty years. However, memory performance has not experienced a gain as dramatic as that of processors; leaving memory performance lagging far behind CPU performance. This enlarging performance gap between processor and memory is referred to as the "memory wall" [1] [2]. The "memory wall" problem exists not only in main memory but also in the on-die caches. For example, in the Intel Nehalem architecture CPU, each L1 data cache has a 4-cycle hit latency; and each L2 cache has a 10-cycle hit latency [3]. Additionally, the IBM Power6 has a 4-cycle L1 cache hit latency, and an L2 cache hit latency of 24 cycles [4]. These large performance gaps between the processor and memory hierarchy make the memory system the dominant performance factor in high-end computing. Intensive research has recently been conducted to improve the performance of memory systems.

Copyright is held by author/owner(s).

However, understanding the performance of modern hierarchical memory systems remains elusive for researchers and practitioners.

Because hierarchical memory systems are a bottleneck of performance, measuring and evaluating memory systems has become an important issue facing the high performance computing community. Conventionally used performance metrics, such as IPC (Instruction Per Cycle) and Flops (Floating point operations per second) are designed from a computing-centric point-of-view, and measure the overall computing performance. They are comprehensive and affected by a multitude of factors such as instruction sets, CPU micro-architecture, memory hierarchy, compiler technologies, etc. and as such are not appropriate measurements of the performance of a memory system. On the other hand, existing memory performance metrics, such as miss rate, bandwidth, and average memory access time (AMAT), only measure a particular component of a memory system. They are useful in optimization and evaluation of a given component, but cannot accurately characterize the performance of the memory system as a whole. In general, a component improvement does not necessarily lead to an improvement in terms of overall computing performance.

There are several reasons that traditional memory performance metrics cannot directly characterize the overall performance of a memory system. First, modern CPUs exploit several ILP (Instruction Level Parallelism) technologies to overlap ALU instruction executions and memory accesses. Out-of-order executions overlap CPU execution time and memory access delay, allowing an application to hide the miss penalty of an L1 data cache miss that hits the L2 cache. Multithreading technology, such as SMT [5] or fine-grained multithreading [6] could tolerate even longer misses through main memory by executing another thread. Speculation mechanisms are used to overcome control dependencies, which help to avoid CPU stalls. Speculation could also activate memory access instructions that are not committed to the CPU registers due to miss predictions. Incorrect speculations can aggravate the burden of data access, but the effect of miss speculations is hard to predict. An incorrect prediction is not always useless. If a wrongly predicted data load accesses the same cache block as the next data load, then the incorrect speculation can be seen as an effective data prefetch, and it benefits the memory performance. Additionally, modern memory systems use a large number of advanced caching technologies to decrease average access latency. Some widely used cache optimization methods, such as non-blocking cache [7], pipelined cache [8], and multibanked cache [9], allow cache accesses to overlap with each other. These technologies make the relationship between memory access and processor performance even more complicated, since the processor could continue accessing memory under multiple cache misses. Thus, the influence of the improvement of one particular component in a memory system becomes increasingly

tangled and elusive. Evaluating memory systems from a single memory access or on a single component does not reflect the complexity of modern memory systems. Advanced memory technologies make the behavior of memory access similar to instruction dispatch, because they both execute several operations concurrently. As with the measurement of instruction executions, memory system evaluations should consider all the underlying parallelism and optimizations to measure the overall performance of a memory system.

A new metric for overall memory system performance, which is separate from CPU performance but at the same time correlates with CPU performance, is needed. In the other words, it should correlate with IPC but measure data access only. The notion of correlating with IPC is important due to the fact that memory performance is a determining factor of the overall performance of modern computing systems. The requirement of separating computing from data access is to provide a better understanding of memory systems, a better understanding of the capacity of a memory system to handle the so-call data-intensive applications, and a better understanding of the match between computing power and memory system performance. To reach this goal, the Access Per Cycle (APC) metric is proposed following the design philosophy of Instructions Per Cycle (IPC). A series of simulation experiments are conducted to confirm that APC is significantly more appropriate than the existing memory metrics. The statistical variable correlation coefficient is used to demonstrate that APC has a 0.97 correlation coefficient value with the overall computing performance in terms of IPC, whereas conventional metrics only have a 0.67 correlation in the best scenarios.

2. INTRODUCTION TO APC

In this section, the formal definition of APC is provided and investigation is carried out on the measurement of memory access cycles of APC in advanced non-blocking memory structures.

2.1 APC Definition

IPC reflects overall performance in terms of the number of executed instructions per cycle. After more than thirty years development, the ILP and memory optimization technologies have many key features in common. Table 1 lists a comparison of some common technologies adopted by them.

Table 1. ILP and memory optimization comparison

ILP Tech.	Memory Tech.	Key feature in common
Pipelined stage in CPU data path	Pipelined Cache	Micro-operation overlapping
Multiple Function Unit	Multiport/Multibanked Cache	Simultaneous operation dispatching
Out-of-order execution	Non-blocking Cache	Do not stall ready operations
Branch prediction/ Speculation/ Runahead[10]	Data Prefetching	Pattern recognizing and only successful with certain possibility

Based on the similarity between processors and memory systems, and inspired by the success of IPC, APC (Access Per Cycle) is proposed to evaluate memory system performance. Generally speaking, APC is measured as the number of memory accesses per cycle. Also, the APC metric can be used to evaluate the memory performance at each level of a memory hierarchy. More specifically, APC is the number of memory accesses requested at a certain memory level (e.g.: L1, L2, L3, or Main Memory) divided by the number of memory access cycles at that level. Let M denote the total data access (load/store) at a certain memory level, and T denote the total cycles consumed by these accesses. According to the definition of APC,

$$APC=MT \qquad (1).$$

The definition is simple enough. However, because modern memory systems adopt many advanced optimizations, such as pipelined cache, non-blocking cache, and multibanked cache, etc. several outstanding memory accesses may co-exist in the memory system at the same time. Counting cycle T is not as simple as it may seem. In the APC definition, it is defined as the total cycle T to be measured based on the *overlapping mode*, which means when there are several memory accesses co-existing during the same cycle, T only increases by one. For memory accesses, the *non-overlapping mode* is adopted. That is all the memory accesses issued are counted, including all successful or non-successful speculations and including all concurrent accesses. For example, if two L1 cache load requests exist at the same time, M will increase by two.

According to the APC definition, each memory level has its own APC value. This paper focuses on APC for L1 cache and includes discussions of the main memory APC. The APC of L1 reflects the overall performance of the memory system, while the main memory's APC is important since it has the longest access latency in the memory hierarchy without considering I/O and file systems. The study of these two should be sufficient in illustrating the APC concept and demonstrating its effectiveness. To avoid confusion, the term APC_D is used for L1 data cache, APC_I is used for L1 instruction cache, which is the number of L1 data or instruction cache accesses divided by the number of overall cache access cycles of their own. Main memory APC is denoted as APC_M, which is the number of off-chip accesses divided by the number of overall main memory access cycles.

2.2 APC Measurement Methodology

As an important cache technology, non-blocking cache can continue supplying data under a certain number of cache misses by adopting Miss Status Holding Register (MSHR) [7]. Calculating an accurate number of overall memory access cycles in a non-blocking cache is not simple. There are two reasons. First, unlike IPC, not every clock cycle has memory access; therefore, we need some form of access detection. Secondly, many different memory accesses can be overlapped. Only one cycle should be counted in the total access cycles even if there are several different memory accesses occurring at the same time. In practice, there could be many different ways to measure the clock cycles under the overlapping mode. In this study, we propose an APC measurement logic, as illustrated in Fig. 1 that supports the overlapping mode to test the potential of APC.

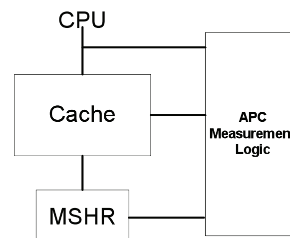


Figure 1. APC Measurement Structure

To avoid overlapping memory accesses from being counted multiple times in an access cycle, the APC Measurement Logic (AML) simultaneously detects memory access activities from CPU, cache and MSHR. If at least one memory access activity exists in the CPU/Cache interface bus or inside the cache, or if outstanding cache miss/misses are registered in the MSHR, this clock cycle is counted as one memory access cycle. Additionally, the AML will count the number of CPU load/store accesses by

detecting CPU/Cache bus requests. If there are several memory requests at the same time from the bus, all are counted. Through this counting, the number of total accesses M and the total memory access cycle T can be obtained, then the APC can be calculated for this level of cache. The pseudocode of memory access counting logic for on-chip caches including L1, L2, or even L3 caches, is shown in Table 2. For L2 and L3 caches, while the logic is the same, multiple buses between upper level caches and itself may be detected simultaneously.

Table 2. Pseudo Code for Memory Access Cycle Counting Logic

```
If(MSHR table is not empty) //Having pending cache miss/misses
        Mem_Cycle ++;
Else if(Cache is accessing)//Cache lookup exists
        Mem_Cycle ++;
Else if(CPU/Cache bus is active)//Having a request or returning data
        Mem_Cycle ++;
Else
      Mem_Cycle does not change
```

The AML for L1 cache can be extended to main memory with little variation. When measuring main memory APC_M, main memory access count and LLC MSHR Cycle are needed to be detected. The former can be found in CPU performance counters; and the latter, only need 1 bit to detect whether the MSHR table is empty or not. If the MSHR is not empty, then the memory cycle should be increased. As a result, there is almost no extra hardware cost to measure the APC_M. Therefore,

$$APC_M = \frac{\text{Main Memory access count}}{\text{LLC MSHR cycles}} \quad (2).$$

3. EXPERIMENTAL SETUP

The motivation of memory evaluation is due to the fact that the final system computing performance is heavily influenced by memory system performance. Therefore, an appropriate memory metric should reflect the system performance. The mathematical statistic variable *correlation coefficient* (CC) is used in this study to determine which memory metric most closely trends with the IPC variation. Correlation coefficient describes the proximity between two variables' changing trends from a statistics viewpoint. It measures how well two variables match with each other. The correlation coefficient is a number between -1 and 1. The higher the CC absolute value is, the closer the relation between the two variables would be [11].

A detailed out-of-order CPU model in the M5 simulator [12] was adopted. Unless stated otherwise, the experiments assume the following processor and cache configuration as default.

Table 3. Simulation Configuration Parameters

Parameter	Value
Processor Function units ROB, LSQ size	1core, 2 GHz, 8-issue width, 6 IntALU 1 cycle, 1 IntMul 3 cycles, 2 FPAdd 2 cycles, 1 FPCmp 2 cycles, 1 FPCvt 2 cycles, 1 FPMul 4 cycles, 1 FPDiv 12 cycles ROB 192, LQ 32, SQ 32
L1 caches	32KB Inst/32KB Data, 2-way, 64B line, hit latency: 2 cycle Inst/2 cycle Data, ICache 10 MSHR Entry, DCache 10 MSHR Entry
L2 cache	2MB, 8-way, 64B line, 12-cycle hit latency, 20 MSHR Entry
DRAM latency/Width	200-cycle access latency/64 bits

Other experimental configurations are based on the default configuration. Each of these only changes one or two parameter/s of the simulation. The detailed experiment configurations are shown in Table 4.

Table 4. A Series of Simulation Configurations

ID	Description	Changed Parameter/s
C1	L1:32KB,2way; L2: 2MB,8way; Mem100ns	Default Config
C2	L1:32KB,4way; L2: 2MB,8way; Mem100ns	L1 Cache Assoc.
C3	L1:32KB,8way; L2: 2MB,8way; Mem100ns	L1 Cache Assoc.
C4	L1:64KB,2way; L2: 2MB,8way; Mem100ns	L1 Cache Size
C5	L1:64KB,4way; L2: 2MB,8way; Mem100ns	L1 Cache Size & Assoc.
C6	L1:64KB,8way; L2: 2MB,8way; Mem100ns	L1 Cache Size & Assoc.
C7	L1:I$32KB,2way, D$64KB,2way; L2: 2MB,8way; Mem100ns	Only DCache Size
C8	L1:I$64KB,2way, D$32KB, 2way; L2: 2MB,8way; Mem100ns	Only ICache Size
C9	L1:I$64KB,4way, D$32KB, 2way; L2: 2MB,8way; Mem100ns	Only ICache Size & Assoc.
C10	L1:I$64KB,8way, D$32KB, 2way; L2: 2MB,8way; Mem100ns	Only ICache Size & Assoc.
C11	L1:32KB,2way; L2: 4MB,8way; Mem100ns	L2 Cache Size
C12	L1:32KB,2way; L2: 8MB,8way; Mem100ns	L2 Cache Size
C13	L1:32KB,2way; L2: 2MB,16way; Mem100ns	L2 Cache Assoc.
C14	L1:32KB,2way; L2: 4MB,16way; Mem100ns	L2 Cache Size & Assoc.
C15	L1:32KB,2way; L2: 8MB,16way; Mem100ns	L2 Cache Size & Assoc.
C16	L1:32KB,2way; L2: 2MB,8way; Mem30ns	Main memory latency
C17	L1:32KB,2way; L2: 2MB,8way; Mem60ns	Main memory latency
C18	L1:32KB,2way, MSHR 1; L2: 2MB,8way; Mem100ns	MSHR Entry
C19	L1:32KB,2way, MSHR 2; L2: 2MB,8way; Mem100ns	MSHR Entry
C20	L1:32KB,2way, MSHR 16; L2: 2MB,8way; Mem100ns	MSHR Entry

The configuration C1~C17 are the basic cache/memory configurations, which only change the cache size, associativity, or memory latency. These basic configurations have 10 MSHR entries each. The C18 changes the cache model into a blocking cache structure. C19 and C20 increase memory level parallelism by increasing the MSHR entry to 2 and 16 respectively. By changing memory system configurations, it is possible to observe memory performance variation trends and IPC variation trends, and examine which memory metric has a performance trend that best matches that of IPC's.

The simulations were conducted with 26 benchmarks from SPEC CPU2006 suite [13]. Five benchmarks in the set were omitted because of compatibility issues with the simulator. The benchmarks were compiled using GCC 4.3.2 with -O2 optimization. The test input sizes provided by the benchmark suite were adopted for all benchmarks. For each benchmark, one billion instructions were simulated to collect statistics, or all executed instructions if the benchmark finished before then.

4. EVALUATION OF RESULTS

Based on the above configurations, the M5 simulator was used to collect the measurements of different memory metrics. Each memory metric is then correlated against the IPC from two approaches. First, based on one configuration, we correlate each application's memory metric with its IPC. Second, we focus on one application, while changing memory configurations, the variation similarity between each memory metric and IPC is observed. The first approach tests the correctness of each memory performance metric. The second tests the sensitivity of each memory performance metric. The combination of these two provides a solid foundation to determine the appropriateness of a metric. The results show that APC has the highest correlation value with IPC in both cases.

4.1 Proximity for different applications

IPC with different memory metrics were compared. The memory metrics compared include, Access per Cycle (APC), Hit Rate (HR, the counterpart of Miss rate), Hits per 1K instruction (HPKI, the counterpart of Misses per 1K instructions), average miss penalty (AMP), and Average Memory Access Time (AMAT) [14]. For APC, HR, and HPKI, there should be a positive relation with IPC; for AMP and AMAT, there should be a negative relation with IPC. To show the proximity of different memory metrics with IPC, Spec CPU2006 was run for all configurations (C1~C20). The correlation coefficient for each memory metric against IPC was calculated and is shown in Fig. 2. From Fig. 2 it can be observed that APC has the strongest relation with IPC, whose average CC value is 0.876. This strong relation between APC and IPC also reflects the fact that the final system performance largely depends on the performance of the memory hierarchy. AMAT is the best, with an average CC value of -0.672 among other metrics, since it considers both hit and miss cases. Compared with AMAT, APC improves correlation value by 30.4%.

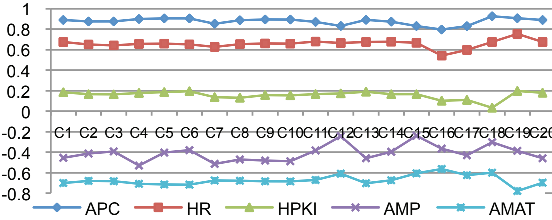


Figure 2. Correlation coefficients under different configurations

4.1.1 *Instruction Cache Affection*

To optimize the accuracy of APC, the L1 instruction cache APC, APC_I is measured. Using APC_D to denote the L1 data cache APC, then APC_{All}, which equals $APC_D \times APC_I$, measures overall L1 performance. The reason APC_D and APC_I are correlated with multiplication is inspired by the conditional possibility. Only when the instruction is efficiently fetched into the CPU, can a data access request be generated.

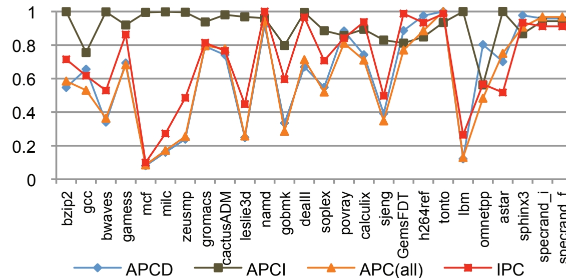


Figure 3. Normalized IPC, APC_I, APC_D, and APC_{All}

The correlation coefficient of APC_{All} under the default configuration (C1) improves by 2.18%. For all configurations (C1~C20), the accuracy of APC_{All} is 0.899, an improvement of 2.58% on average compared to APC_D. For instruction accesses, most applications work well with little catch misses. Only very small numbers of applications, such as gcc, GemsFDTD, and h264ref, have large instruction miss rates. For these applications, APC_{All} is a more accurate measurement. The normalized APC_I, APC_D, APC_{All} and IPC of default configuration (C1) are shown in Fig. 3. The normalization is based on the magnitude of each metric. From Fig. 3, it can be seen that the APC_{All} and IPC have almost the same variation trends. Fig. 3 confirms that the overall application performance is largely determined by the memory performance as can be expressed by the APC metric.

4.2 Proximity for different configurations

In this section, we focus on each application, in order to observe the impact of memory configuration on performance and correlation. Two groups of 20 simulation configurations are measured. The first group (C1~C17 in Table 4) changes basic cache/memory configuration. Each configuration inside the first group changes L1/L2 cache size or associativity, or memory latency respectively. The other group (C18~C20) changes non-blocking cache ability with altering the number of MSHR entry. The simulation results of the first group are presented in this section. In the next section, simulations with both groups are conducted. To clearly show the difference, APC_D and APC_{All} are compared against the other four memory metrics from Fig. 4 to Fig. 7. Similar to APC which has two measures APC_D and APC_{All}, every conventional memory metric also has two measures which represent data cache performance only and comprehensive cache performance (data and instruction cache combined performance). For example, when considering AMAT, $AMAT_D$ is used to describe data cache AMAT, and $AMAT_{All}$ (equal to $AMAT_D \times AMAT_I$) describes comprehensive cache AMAT.

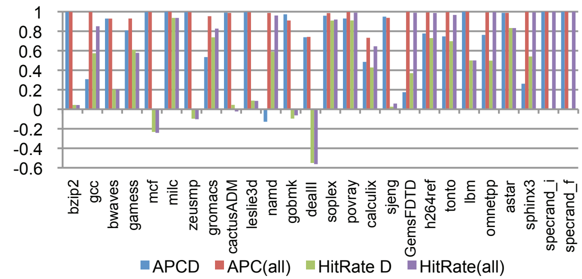


Figure 4. The Correlation Coefficients of APC and HR

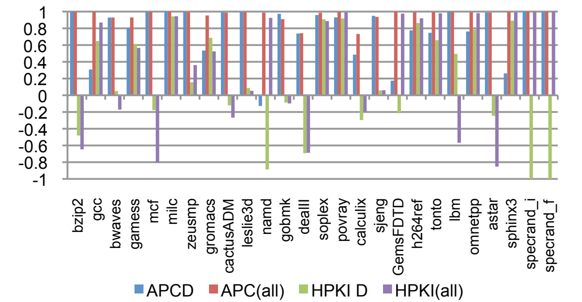


Figure 5. The Correlation Coefficients of APC and HPKI

In Fig. 4 to Fig. 7, it can be seen that APC_{All} has the highest correlation coefficient value with IPC, with an average value for all applications of 0.9632, and this means that APC_{All} and IPC have a dominant relation. For other memory metrics, $AMAT_{All}$ has the closest relation with IPC, with an average value of -0.9393, a little lower than APC_{All}. This shows if advanced data access technologies, such as non-blocking, are not considered, AMAT is a quite good metric in reflecting memory performance variation. However, when considering non-blocking structure, AMAT is misleading. (Please refer to next section for details). For other metrics, there are some misleading indications as well. For example, Hit Rate should have positive coefficient values, but for several applications its coefficient values are negative or approximate to zero.

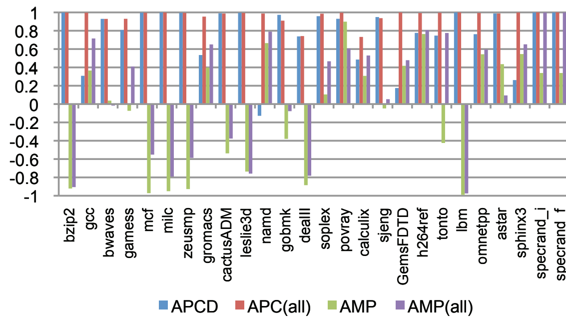


Figure 6. The Correlation Coefficients of APC and AMP

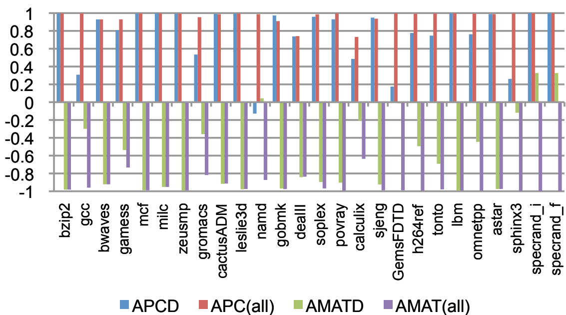


Figure 7. The Correlation Coefficients of APC and AMAT

One reason for the divergence is that HR does not explicitly include lower level cache performance factors, such as miss latency information (whereas APC and AMAT do consider lower level cache performance), so when L2 cache size increases or main memory latency decreases, the IPC will increase, but Hit Rate does not change. AMP only considers lower level cache miss access. When L1 data/instruction cache size increases or associativity increases, the number of miss accesses decreases, but the miss latency for each miss may not change.

4.3 Changing Cache Access Parallelism

APC and AMAT have very similar correlation values with IPC when changing basic cache/memory configurations. Here the number of MSHR entries is altered to examine whether the two memory metrics still have a similar relation with IPC. Fig. 8 shows the correlation coefficient of APC and AMAT for all the twenty configurations listed in Table 4.

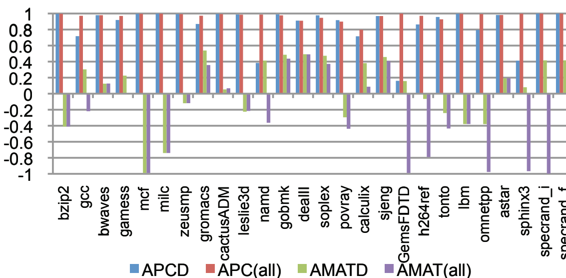


Figure 8. The Correlation Coefficients of APC and AMAT with MSHR changes

Fig. 8 shows that APC still has the same correlation, with an average value of 0.9696. However, AMAT could not correctly reflect IPC for most applications. The reason is that when there are not enough MSHR entries, the CPU will be blocked by the memory system. APC can record the CPU blocked cycles by detecting MSHR's non-emptiness, whereas AMAT cannot. On the contrary, contention increases with the number of MSHR entries; therefore, the individual data access time, that is AMAT, will increase as well. The variation of IPC and APC matches well with each other. However, AMAT gives a false indication about memory system performance.

Through the above simulation and analyses, under different applications and under different configurations, it can be seen that the APC metric is the most appropriate performance metric for memory systems. APC can directly determine the overall system performance in all the testing. In contrast, other existing memory metrics cannot accurately reflect the system performance, and sometimes even mislead the performance.

5. DISCUSSION

5.1 Bottleneck inside memory system

An important question is at which level of a memory system is the actual bottleneck of its performance. According to the APC's definition, each level of memory hierarchy has its own APC values: L1 data and instruction caches have APC_D and APC_I respectively; L2 cache has APC_{L2}; and main memory has APC_M. However, each level's APC not only represents the performance of its memory level, but also includes all the lower levels of the memory hierarchy. For example, the value of APC_D represents the memory performance of L1 data cache, L2 cache and main memory; and APC_{L2} represents the memory performance of L2 cache and main memory. Only APC_M, which is the lowest level in the memory hierarchy, represents main memory itself when disk storage is not considered. Therefore, by correlating IPC with APC at each level, one can find the lowest level that has a dominating correlation with IPC and can quantitatively detect the performance bottleneck inside the memory system. Fig. 9 shows the correlation value of all level APCs for 26 benchmarks. For example, for benchmark *mcf*, both APC_{All} and APC_{L2} have dominant relation with IPC, but its APC_M does not. That means the performance of *mcf* application is determined by its L2 cache performance, and has a good locality. For benchmark *lbm*, since all levels of APC have dominating correlation with IPC, the performance of *lbm* is determined by the performance of the lowest level, namely the main memory level.

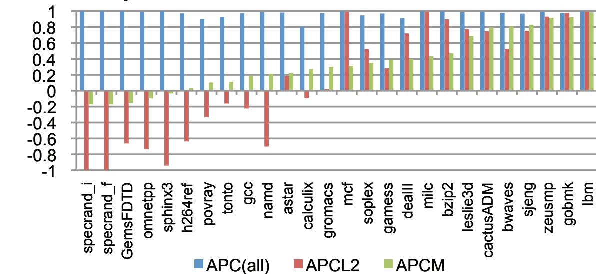


Figure 9. Correlation Coefficients of APC_{All}, APC_{L2}, and APC_M

5.2 A quantitative definition of data intensive

The term "*Data-intensive Applications*" and "Data-intensive Computing" are widely-used terms in describing application or computing where data movement, instead of computing, is the dominant factor. However, there is no commonly accepted definition of "data-intensive computing" or quantitative measurement of "data intensive". As APC characterizes the overall memory performance, the IPC and APC correlation value provides a quantitative definition of data intensive. The idea is simple: if APC_M dominates IPC performance, then the application is data intensive. The degree of dominance provides a

measurement of data intensiveness. We use the correlation value of APC_M to quantify the degree of data intensiveness. There are three reasons to use APC_M instead of APC_D to measure data intensiveness. First, due to the "memory-wall" problem, memory latency becomes the most important performance bottleneck in the memory hierarchy and dramatically drags CPU speed. Next, we do not count data re-use as part of data-intensiveness unless it has to be read from main memory again. Finally, according to the definitions of APCs, if the APC_M has a dominant relation with IPC, then APC_{L2} and APC_D will also have a dominant relation with IPC.

Fig. 9 is sorted according to APC_M correlation values in ascending order (the farther to the left, the smaller the value of APC_M). The correlation value of APC_M is divided into three intervals, that is (-1, 0.3), [0.3, 0.9), and [0.9, 1). Thirteen applications counted from the left side (from *specrand_i* to *gromacs*) fall into the first interval. According to the three APC values used in Figure 9, it can be concluded that the application performance of these 13 applications are dominated by the L1 cache, not L2 or main memory because the correlation values of APC_{L2} and APC_M of these applications are negative or very small, As the correlation value of APC_M increases, the effect of main memory to the overall application becomes increasingly important. Therefore, in the second interval (from *mcf* to *sjeng*), some applications' performance are dominated by the L2 caches, e.g. *mcf, milc*. However, for some other applications, such as *bzip2*, L2 and main memory are both important. For the third interval, the applications' performances are dominated by main memory performance. This observation motivates us to define an application data intensive if its correlation coefficient of APC_M and IPC is equal to or larger than 0.9. Another reason for picking 0.9 as the threshold is that, according to mathematical definition of correlation coefficient, when the correlation value of two variables is equal to or larger than 0.9, then the two variables have a dominant relation. Therefore, here we define that an application is data intensive *if and only if*

$$\text{Data Intensive Application} \equiv coe\,APCM, IPC \geq 0.9,$$

and the value of the correlation provides a quantitative measurement of the data-intensiveness.

6. RELATED WORK

Except the four traditional memory metrics (miss rate (MR), miss per kilo-instructions (MPKI), average miss penalty (AMP), and average memory access time (AMAT) [14]) listed in the earlier sections, there is a new main memory metric called Memory Level Parallelism (MLP) [15]. It is the average number of long-latency main memory outstanding accesses when there is at least one such outstanding access [16]. Assuming each off-chip memory access has a constant latency, say *m* cycles, one can prove $APC_M = MLP/m$. That means APC_M is directly proportional to MLP. Any analyzing indication from MLP on CPU micro-architecture could also be obtained from APC_M. A known limitation of MLP is it only focuses on off-chip memory access based on the epoch memory access mode for some commercial or database workloads [16]. For some traditional CPU intensive applications, only considering main memory access is far from enough. In contrast, APC not only can be used to analyze commercial applications, but also to analyze traditional scientific applications, so it has a much wider application.

7. CONCLUSION

In this paper we proposed a new memory metric, APC, gave its measurement methodology, and demonstrated its unique ability in measuring the overall and layered performance of modern hierarchical memory systems. Intensive simulations were conducted with a modern computer system simulator, M5, to verify the potential of APC and compared it with existing memory performance metrics. Simulation and statistical results show that APC is a significantly more appropriate memory metric than other existing memory metrics when reflecting overall performance of a memory system. APC can be applied at different levels of a memory hierarchy. Based on the correlation coefficient with different level APCs and IPC, the bottleneck of a memory hierarchy can be identified. In addition, the correlation of APC and IPC provides a quantitative measurement of data-intensiveness of an application. It provides a measurement of data-centric computing, and could have profound impact in future data-centric algorithm design and system development.

8. REFERENCES

[1] X.-H. Sun, and L. Ni, Another View on Parallel Speedup, *Proc. of IEEE Supercomputing'90*, NY, Nov. 1990.

[2] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. ACM SIGArch Computer Architecture News, Mar. 1995.

[3] Michael E. Thomadakis, The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, A research report of Texas A&M University, Mar. 2011. http://sc.tamu.edu/systems/eos / nehalem.pdf

[4] Robert Fiedler, Blue Waters Architecture, Great lakes consortium for Petascale Computation. Oct. 2010.

[5] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, et.al Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor, *in Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996, 191~202.

[6] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun, Niagara: A 32-Way Multithreaded SPARC Processor, *IEEE Micro*, 21-29, Mar. 2005.

[7] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. *In 8th International Symposium on Computer Architecture* (ISCA81), 81–87, 1981.

[8] Amit Agarwal, Kaushik Roy, T. N. Vijaykumar: Exploring High Bandwidth Pipelined Cache Architecture for Scaled Technology. 2003: 10778-10783

[9] Jude A. Rivers, Gary S. Tyson, Edward S. Davidson, et.al, On High-Bandwidth Data Cache Design for Multi-Issue Processors, *IEEE Micro-30*, Dec. 1997.

[10] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt, Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors, *in Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, 129-140, Anaheim, CA, Feb. 2003.

[11] Jim Higgins, The Radical Statistician: A Practical Guide to Unleashing the Power of Applied Statistics in the Real World, Biddle Consulting Group, Apr. 2011. http://www.biddle.com/documents/bcg_comp_chapter2.pdf,

[12] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro, 26(4):52*, Jul. 2006.

[13] C. D. Spradling, SPEC CPU2006 benchmark tools. *ACM SIGARCH Computer Architecture News*, 2007.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, Sep. 2006.

[15] A. Glew, "MLP yes! ILP no!," in ASPLOS Wild and Crazy Idea Session '98, Oct. 1998.

[16] Y. Chou, B. Fahs and S. Abraham, Microarchitecture Optimizations for Exploiting Memory-Level Parallelism, *in 31st International Symposium on Computer Architecture* (ISCA04), Jun. 2004.

HOTL: a Higher Order Theory of Locality

Xiaoya Xiang Chen Ding Hao Luo

Department of Computer Science
University of Rochester
{xiang, cding, hluo}@cs.rochester.edu

Bin Bao *

Adobe Systems Incorporated
bbao@adobe.com

Abstract

The locality metrics are many, for example, miss ratio to test performance, data footprint to manage cache sharing, and reuse distance to analyze and optimize a program. It is unclear how different metrics are related, whether one subsumes another, and what combination may represent locality completely.

This paper first derives a set of formulas to convert between five locality metrics and gives the condition for correctness. The transformation is analogous to differentiation and integration. As a result, these metrics can be assigned an order and organized into a hierarchy.

Using the new theory, the paper then develops two techniques: one measures the locality in real time without special hardware support, and the other predicts multicore cache interference without parallel testing. The paper evaluates them using sequential and parallel programs as well as for a parallel mix of sequential programs.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Modeling Techniques

General Terms Measurement, Performance, Theory

Keywords Locality metrics, Locality modeling

1. Introduction

The memory system of a computer is organized as a hierarchy. Locality metrics are used in software and hardware to manage and optimize the use of the memory hierarchy. For locality analysis, the basic unit of information is a data access, and the basic relation is a data reuse. The theory of locality is concerned with the fundamental properties of data accesses and reuses, just as the graph theory is with nodes and their links.

An influential theory developed over the past four decades is the working-set locality theory (WSLT) [14]. In this paper, we develop a similar theory for cache locality (CLT). Cache locality metrics are many and varied. To quantify performance, we use the miss rate. To manage sharing, we use the footprint. To analyze and optimize a program, we use the reuse distance. Some metrics are hardware dependent, useful for evaluating a specific machine and managing it at run time. Others are hardware independent, useful for optimizing a program for all cache sizes. The two types of metrics are converging in multicore caching, where the total cache size is fixed but the available portion for each program varies.

In this paper we consider five locality metrics, with a short description here and the precise definitions in the next section.

- *Footprint*: the expected amount of data a program accesses in a given length window.
- *Inter-miss time*: the average time between two cache misses in a given size cache.
- *Volume fill time*: the average time for the program to access a given volume of data.
- *Miss ratio*: the fraction of references that cause cache misses.
- *Reuse distance*: for each data access, the amount of data accessed between this and the previous access to the same datum.

To denote them collectively, we insert 'et' between the last two, take the initial letters (except for the fill time from which we take one 'l'), and produce the acronym "Filmer".

We present a theory showing that the five Filmer metrics can be mutually derived from each other. The conversion involves taking the difference in one direction and the sum in the reverse direction. The theoretical relation is analogous to differentiation and integration. Hence we call it a *higher order theory* of locality (HOTL).

Similar conversions have been part of the working set theory, making it the first HOTL theory (Section 2.8). The working set theory was developed to analyze locality in the main memory. The new theory we develop is for cache memory. It endows each of the five cache locality metrics the collective strength of all its Filmer peers:

- *Efficiency*. If we can measure one Filmer metric on-line, we can calculate all the others at the same time.
- *Composability*. The miss rate does not compose in that when a group of programs are run together, the number of misses is not the sum of the misses of each member running alone. If another Filmer metric is composable, then we can compose the miss rate indirectly.
- *Hardware sensitivity*. If we can measure the effect of cache associativity and other hardware parameters on the miss rate, we can compute their impact on the other metrics.

The conversion methods we describe are not always accurate. The correctness depends on whether the footprint statistics in reuse windows is similar to the footprint in general windows, in other words, whether the reuse windows are representative of general windows. We call the condition the *reuse-window hypothesis*. The Filmer metrics capture different aspects of an execution: the reuse distance is per access, the footprint is per window, while the miss-

* The work was done when Bin Bao was a graduate student at the University of Rochester.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

Copyright © 2013 ACM 978-1-4503-1870-9/13/03...$10.00

ratio has the characteristics of both. Their conversion creates conflicts, and the reuse-window hypothesis is the condition for reconciliation.

Our recent work shows that one of the Filmer metrics, the average data footprint, can be computed efficiently [46]. In this work, we further improve the efficiency through sampling. More importantly, we apply the HOTL theory to convert it to reuse distance and predict the miss ratio. The purpose of the miss-ratio prediction is twofold: to validate the theory and to show a practical value. The main results are:

- *Real-time locality measurement.* The HOTL-enabled technique predicts the miss ratio for thousands of cache sizes with a negligible overhead. When tested on SPEC 2006 and PARSEC parallel benchmarks, the prediction matches the actual miss ratio measured using the hardware counters. Without sampling, the analysis is 39% faster than simulating a single cache size. With sampling, the end-to-end slowdown is less than 0.5% on average with only three programs over 1%.
- *Cache interference prediction.* The HOTL-enabled technique predicts the effect of cache sharing without parallel testing. For pair interference, the result can be characterized as half-and-half (Section 4.5).

Knowing the miss rate does not mean knowing the memory performance. The actual effect of a cache miss depends significantly on data prefetching, memory-bus arbitration, and other factors either in the CPU above the cache hierarchy or the main memory below. In this paper, we limit our scope to the models of data and cache usage and to methods that measure and reduce the number of cache misses.

2. The Higher Order Theory of Cache Locality

The theory includes a series of conversion methods and their correctness condition. We will refer to these methods collectively as the HOTL conversion for the Filmer metrics.

2.1 Locality Metrics

The working set theory defines the locality metrics to measure the intrinsic demand of a process [13]. The actual performance is the hardware response to the program demand. By defining locality metrics independent of their specific uses, the approach combines clarity and concision on the one hand and usefulness and flexibility on the other. We follow the same approach and say that a locality metric is *program intrinsic* if it uses only the information from the data access trace of a program. Throughout the paper, we use n to denote the length of the trace and m the total amount of data accessed in the trace.

A footprint is defined on a time window, and the miss ratio for a cache size. Since we do not know a priori in which window or cache the metrics may be used, we define the footprint and miss ratio metrics to include all windows and all cache sizes — they are functions over a parameter range.

The five metrics we consider are program intrinsic functions defined on a sequential data access trace. The time is logical and counted by the number of data accesses from the start of the execution. The cache is fully associative and uses the LRU replacement, with a fixed cache-block size. We will consider the physical time and set associative cache when we apply the basic theory. We use the term miss ratio if the time is logical and miss rate if it is physical.

2.2 Average Footprint

A footprint is the amount of data accessed in a time window. A performance tool often measures it for some execution window, i.e. taking a snapshot. A complete measure should consider *all* execution windows. For each length l, the average footprint $fp(l)$ is the average footprint size in all windows of length l.

Let W be the set of all length-l windows in a length-n trace. Each window w has a footprint fp_w. The average footprint $fp(l)$ is the total footprint in these windows divided by $n - l + 1$, the number of the length-l windows.

$$fp(l) = \frac{\sum_{all\ w\ of\ length\ l} fp_w}{n - l + 1}$$

For example, the trace "abbb" has 3 windows of length 2: "ab", "bb", and "bb". The size of the 3 footprints is 2, 1, and 1, so $fp(2) = (2 + 1 + 1)/3 = 4/3$.

The footprint is composable in that the combined footprint of two programs is the sum of their individual footprints (assuming no data sharing). We have used this property when developing efficient models of cache sharing [45, 46]. Another useful property, which we will explore in Section 3, is that the footprint is amenable to sampling.

The working set theory defined the average number of pages accessed in a time window as the working set size and gave a linear-time method to estimate the size [13]. A number of other approximate solutions followed [9, 27, 36, 39]. Our recent work gave two algorithms to measure the footprints in all execution windows and compute either the distribution [45] or the average [46] of the footprints for windows of the same length. The average footprint, e.g. the one in the preceding example, can be computed precisely in linear time. We use the average footprint in this work. Our measurement algorithm [46] will play a critical role in the new theory in Section 2.7.

2.3 Volume Fill Time

Intuitively, we may consider the cache as a reservoir and the data access of a program a stream feeding into the reservoir with new content. Having a fixed capacity, the reservoir discharges (evicts) previous volumes as it receives the new flows. The key concept in this analogy is the volume fill time, the time taken for a stream to fill the reservoir.

The volume fill time is the time a program takes to access a given amount of data, or symbolically, $vt(v)$ for volume v. The metric is program intrinsic. To model hardware, we simplify and assume that the cache is fully associative LRU. Under the assumption, the volume fill time $vt(c)$ is the time for a program to fill the cache of size c. Whether the cache is empty or not, after $vt(c)$, the cache is populated with the data (and only the data) accessed in the last $vt(c)$ time. In the cold-start cache, all data will be brought in by cache misses. In the warm cache, the fraction of the data already in the cache will stay, and the rest will be brought in by cache misses. We call the volume fill time interchangeably as the *cache fill time*.

The fill time can be defined in two different ways. First, we define it as the inverse of the footprint function:

$$vt(c) = \begin{cases} fp^{-1}(c) & \text{if } 0 \le c \le m \\ \infty & \text{if } c > m \end{cases}$$

where m is the total amount of program data. Within the range $0 \le c \le m$, the invariant $fp(vt(c)) = fp(fp^{-1}(c)) = c$ symbolizes the conversion that when the footprint is the cache size, the footprint window is the fill time. The conversion is shown visually in Figure 1. From the average footprint curve, we find the cache size c on the y-axis and draw a level line to the right. At the point the line meets the curve, the x-axis value is the fill time $vt(c)$.

A careful reader may question the uniqueness of the fill time. For example for the trace "xx...x", it is unclear what should be the fill time $vt(1)$. When defined as the inverse function fp^{-1},

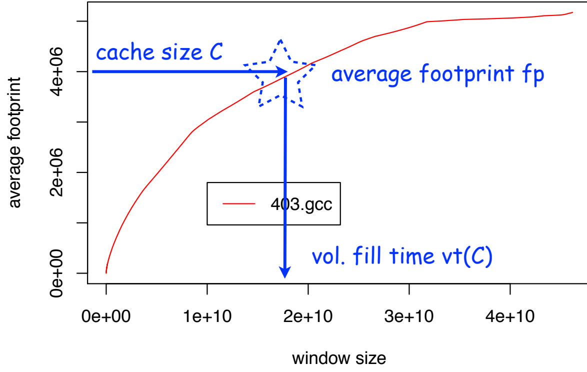


Figure 1: Defining the volume fill time using the footprint.

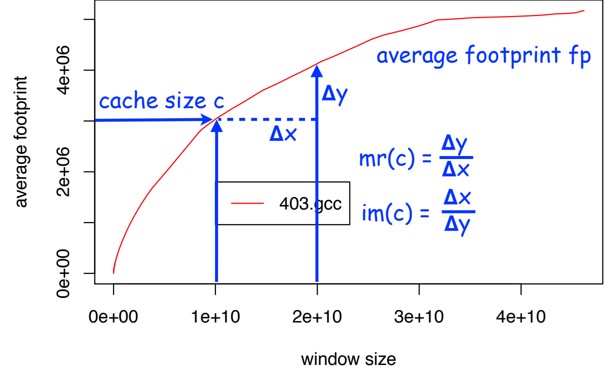


Figure 2: Equivalent conversions of the footprint to the miss ratio and the fill time to the inter-miss time.

the same problem happens if there are x_1, x_2 such that $fp(x_1) = fp(x_2)$. However, this problem does not occur using the footprint-based definition. We will prove later in Section 2.7 that the average footprint is a concave function. As a result, it is strictly increasing, and as its inverse, vt is a proper function and strictly increasing as well. We call the footprint-based definition the *Filmer fill time*.

Alternatively, we can define the fill time in a different way. For the volume v, we find all windows in which the program accesses v amount of data. The average window length is then the fill time. We refer to the second definition the *direct fill time*, since it is defined directly, not through function inversion.

Consider another example trace "abbc". The Filmer fill time is $vt_{Filmer}(1) = 1$, since all single-element windows access one datum. The direct fill time takes the 5 windows with the unit-size data access: "a", "b","b", "bb", and "c" and computes the average $vt_{direct}(1) = (1+1+1+2+1)/5 = 6/5$. The Filmer definition uses the windows of the same length. The direct definition uses the windows of possibly different lengths.

The cache fill time is related to the residence time in the working set theory [14]. Once a program accesses in a data block but stops using it afterwards, its residence time in cache is the time it stays in cache before being evicted.

In Appendix A, we give an algorithm to measure the direct fill time. In Section 4.4, we show that the direct definition has serious flaws and is unusable in practice. Unless explicitly specified in the rest of the paper, by fill time we mean the Filmer fill time.

2.4 Inter-miss Time and Miss Ratio

We derive the inter-miss time for fully associative LRU cache of size c. Starting at a random spot in an execution, run for time $vt(c)$, the program accesses c amount of data and populates the cache of size c. It continues to run and use the data in the cache until the time $vt(c+1)$, when a new data block is accessed, triggering a capacity or a compulsory miss [24]. The time interval, $vt(c+1) - vt(c)$, is the miss-free period when the program uses only the data in cache. We use this interval as the average inter-miss time $im(c)$[1]. The reciprocal of $im(c)$ is the miss ratio $mr(c)$.

$$im(c) = \begin{cases} vt(c+1) - vt(c) & \text{if } 0 \le c < m \\ \frac{n}{m} & \text{if } c \ge m \end{cases}$$

Since the fill time is the inverse function of the footprint, we can compute the miss ratio from the footprint directly. The direct conversion is simpler and more efficient. In practice, we measure the footprint not for all window sizes but only those in a logarithmic series. Let x and $x + \Delta x$ be two consecutive window sizes we measure, we then compute the miss ratio for cache size $c = fp(x)$:

$$mr(c) = mr(fp(x)) = \frac{fp(x + \Delta x) - fp(x)}{\Delta x}$$

Being a simpler and more general formula, we will use it in the theoretical analysis and empirical evaluation. To cover all cache sizes in practice, we use it as the miss ratio for all cache sizes $c \in [fp(x), fp(x + \Delta x))$.

The fill time (vt) conversion and the footprint (fp) conversion are equivalent. Figure 2 shows the two visually. For the same two data points on the footprint curve, let $\Delta x = x_2 - x_1$ be the difference in the window length and $\Delta y = y_2 - y_1$ be the difference in the amount of data access. The fill time conversion computes the inter-miss time $im(y_1) = \frac{vt(y_2)-vt(y_1)}{y_2-y_1} = \frac{\Delta x}{\Delta y}$, and the footprint conversion computes the miss ratio $mr(fp(x_1)) = mr(y_1) = \frac{fp(x_2)-fp(x_1)}{x_2-x_1} = \frac{\Delta y}{\Delta x}$.

For associative cache, Smith showed that cache conflicts can be estimated based on the reuse distance [37]. Hill and Smith evaluated how closely such estimate matched with the result of cache simulation [25]. We next derive the reuse distance. Once derived, we can use it and the Smith formula to estimate the effect of cache conflicts and refine the miss ratio prediction.

2.5 Reuse Distance

For each memory access, the *reuse distance*, or *LRU stack distance*, is the number of distinct data used between this and the previous access to the same datum [31]. The reuse distance includes the datum itself, so it is at least 1. The probability function $P(rd = c)$ gives the fraction of data accesses that have the reuse distance c. The capacity miss ratio, $mr(c)$, is the total fraction of reuse distances greater than the cache size c, i.e. $mr(c) = P(rd > c)$. Consequently,

$$P(rd = c) = mr(c - 1) - mr(c)$$

The reuse distance has extensive uses in program analysis and locality optimization. Any transformation that shortens a long reuse distance reduces the chance of a cache miss. At the program level, reuse distance analysis extends dependence analysis, which identifies reuses of program data [1], to count the volume of the intervening data [4, 8, 10]. At the trace level, the analysis can correlate the change in locality in different runs to derive program-level patterns and complement static analysis [21, 30, 49].

[1] In the working-set theory, the corresponding metric is the time between page faults and known as the lifetime.

To review the conversion formulas, let's consider the example trace "xyzxyz...". Assuming it infinitely repeating, we have $m = 3$ and $n = \infty$. The following table shows the discrete values of the Filmer metrics computed according to the HOTL conversion.

t	$fp(t)$	c	$vt(c)$	$im(c)$	$mr(c)$	P(rd=c)
1	1	1	1	1	1	0
2	2	2	2	1	1	0
3	3	3	3	∞	0	1
4	3	4	∞	∞	0	0

2.6 The Higher Order Relations

In algebra, the term order may refer to the degree of a polynomial. Through differentiation, a higher order function can derive a lower order function. If we use the concept liberally on locality functions (over the discrete integer domain), we see a higher order locality theory, as shown in a metrics hierarchy in Figure 3.

locality metrics	formal property	useful characteristics
3rd order: footprint, volume fill time	concave/ convex	linear-time, amenable to sampling, composable (dynamic locality)
2nd order: miss ratio, inter-miss time	monotone	machine specific, e.g. cache size/associativity (cache locality)
1st order: reuse distance	non-negative	decomposable by code units and data structures (program locality)

Figure 3: The hierarchy of cache locality metrics. The five locality metrics are mutually derivable by either taking the difference of the metrics when moving down the hierarchy or taking the sum of the metrics when moving up.

In the preceding sections, we have shown the series of conversions from the third order metric, the footprint, to the first order metric, the reuse distance. To compute a lower order metric, the HOTL conversion takes the difference of the function of a higher order metric. The inter-miss time is the difference of the fill times, and the reuse distance is the difference of the miss ratios.

The conversion formulas are all reversible. We can calculate a higher order metric by integrating the function of a lower order metric. For example, the miss ratio is the sum of the reuse distances greater than the cache size. The fill time is the sum of the inter-miss times up to the cache size.

The mathematical property is different depending on the order of the locality metric, as shown in the second column in Figure 3. Going bottom up, the reuse distance is a distribution, so the range is non-negative. For just compulsory and capacity misses, the miss ratio is monotone and non-increasing, i.e. the stack property [31]. The footprint has been shown to be monotone [46]. Later we will prove a stronger property.

Although the phrase higher order was not used, the working set theory was about the higher order relations between the working set size, the miss rate, and the reuse-time interval. In Section 2.8, we will compare the two higher order theories.

2.7 The Correctness Condition

The conversion from the footprint to the miss ratio is not always correct. To understand correctness, consider the reuse distance and the footprint both as window statistics. The reuse distance is the footprint of a reuse window. A reuse window starts and finishes with two accesses to the same datum with no intervening reuses. For a program with n accesses to m data, there are $n - m$ finite-length reuse windows. They are a subset of all windows. The number of all windows is n choose 2 or $\frac{n(n+1)}{2}$. We define the average footprint over all reuse windows as $rfp(l)$, the same way we define $fp(l)$ over all windows.

In this section, we show the correctness condition: for the HOTL conversions to be correct, the two functions, $fp(l)$ and $rfp(l)$, must be equal.

To show this result, we introduce a different formula for predicting the miss ratio. To estimate whether an access is a miss for cache size c, we take the reuse window length l, find the average footprint $fp(l)$, and predict it a cache miss if and only if $fp(l) > c$. We call this method the *reuse-time conversion*. Let $P(rt = t)$ be the density function of the reuse time, that is, the fraction of reuse windows with the length t. The miss ratio predicted by the reuse-time conversion is as follows. We label the result mr_{rt} to indicate that the prediction is based on the reuse time. The first access to a datum has the reuse time of ∞.

$$mr_{rt}(fp(l)) = P(rt > l) = \sum_{t=l+1}^{\infty} P(rt = t)$$

If we re-label $fp(l)$ as the working set size, the formula is identical to that of Denning and Schwartz (Section 2.8). However, the use of $fp(l)$ is an important difference. The reuse-time conversion is a modified version of Denning and Schwartz. We may call it an augmented Denning-Schwartz conversion.

Take the example trace "xxyxxz". Two of the average footprints are $fp(3) = 2$ and $fp(4) = \frac{7}{3}$. The reuse times, i.e. the length of the reuse windows, are $\infty, 2, \infty, 3, 2, \infty$. The reuse-time conversion is $mr_{rt}(2) = mr_{rt}(fp(3)) = \sum_{t=4}^{\infty} P(rt = t) = 50\%$. The Filmer conversion is based on the footprint. We call it mr_{fp} and have $mr_{fp}(2) = fp(4) - fp(3) = 33\%$. In general for small traces, the reuse-time conversion is more accurate, as is the case in this example.

Next we prove that for large traces, the miss ratio prediction is the same whether using the reuse time or using the footprint. Then we will show the correctness condition of the entire HOTL theory as a corollary.

From the view of the locality-metrics hierarchy, the reuse-time conversion is bottom up from a first-order metric to a second-order metric. The footprint conversion is top-down from a third-order metric to the same second-order metric. If they meet and produce the same result, we have the equivalence relation across the entire hierarchy.

To prove the equivalence, we need the recently published formula that computes the average footprint from the reuse-time distribution [46].

Lemma 2.1 (Xiang formula [46]).

$$\begin{aligned} fp(w) &= m - \frac{1}{n-w+1}\Big(\sum_{i=1}^{m}(f_i - w)I(f_i > w) \\ &+ \sum_{i=1}^{m}(l_i - w)I(l_i > w) \\ &+ n\sum_{t=w+1}^{n-1}(t - w)P(rt = t)\Big) \end{aligned} \quad (1)$$

The symbols are defined as:

- f_i: the first access time of the i-th datum.

- l_i: the *reverse* last access time of the i-th datum. If the last access is at position x, $l_i = n + 1 - x$, that is, the first access time in the reverse trace.
- $P(rt = t)$: the fraction of accesses with a reuse time t.
- $I(p)$: the predicate function equals to 1 if p is true; otherwise 0.

If we assume $n \gg w$, the equation can be simplified to

$$fp(w) \approx m - \sum_{t=w+1}^{n-1} (t - w)P(rt = t)$$

Theorem 2.2 (Footprint and reuse-time conversion equivalence). *For long executions ($n \gg w$), the footprint conversion and the reuse-time conversion produce equivalent miss-ratio predictions.*

Proof Let the cache size be c and l and $l + x$ be two consecutive window sizes such that $c \in [fp(l), fp(l + x))$. The miss ratio by the footprint conversion is $\frac{fp(l+x)-fp(l)}{x}$.

Expand the numerator $fp(l + x) - fp(l)$ using the approximate equation from Lemma 2.1:

$$
\begin{aligned}
& fp(l + x) - fp(l) \\
\approx & m - \sum_{t=l+x+1}^{n-1} (t - l - x)P(rt = t) - m + \sum_{t=l+1}^{n-1} (t - l)P(rt = t) \\
= & \sum_{t=l+1}^{n-1} (t - l)P(rt = t) - \sum_{t=l+x+1}^{n-1} (t - l - x)P(rt = t) \\
= & \sum_{t=l+1}^{l+x} (t - l)P(rt = t) + \sum_{t=l+x+1}^{n-1} (t - l)P(rt = t) \\
& - \sum_{t=l+x+1}^{n-1} (t - l - x)P(rt = t) \\
= & \sum_{t=l+1}^{l+x} (t - l)P(rt = t) + x \sum_{t=l+x+1}^{n-1} P(rt = t) \\
\approx & \sum_{t=l+1}^{l+x} xP(rt = t) + x \sum_{t=l+x+1}^{n-1} P(rt = t) \\
= & x \sum_{t=l+1}^{n-1} P(rt = t) \\
\approx & x \sum_{t=l+1}^{\infty} P(rt = t)
\end{aligned}
$$

The miss ratio, $\frac{fp(l+x)-fp(l)}{x}$, is approximately $\sum_{t=l+1}^{\infty} P(rt = t)$, which is the result of the reuse-time conversion. Note that the equation is approximately true also because of the earlier simplifications made to the Xiang formula. ∎

The two predictions being the same does not mean that they are correct. They may be both wrong. Since the correct calculation can be done using reuse distance, the correctness would follow if from the reuse time, we can produce reuse distance. In other words, the correctness depends on whether the all-window footprint used by the reuse time conversion is indeed the reuse distance. We can phrase the correctness condition as follows:

COROLLARY 2.3 (Correctness). *The footprint-based conversions are accurate if the footprints in all reuse windows have the same distribution as the footprints in all windows, for every reuse window length l.*

When the two are equal, using the all-window footprint is the same as using the reuse distance. We posit as a hypothesis that the condition holds in practice, so the HOTL conversion is accurate. We call it the *reuse-window hypothesis*.

Consider the following two traces. The second trace has a smaller difference between the all-window footprint fp and the reuse-window footprint rfp. The smaller difference leads to more accurate miss ratio prediction by HOTL. The hypothesis does not hold in either trace, so the prediction is not completely accurate. As to real applications, we will show an empirical evaluation for the full suite of SPEC CPU2006 benchmark programs [23] and a number of PARSEC parallel programs [6].

trace	$fp(2)$	$rfp(2)$	mr(1)		error
			pred	*real*	$\|pred - real\|$
wwwx	4/3	1	1/3	2/4	17%
wwwwx	5/4	1	1/4	2/5	5%

Finally, we show another consequence of Theorem 2.2.

COROLLARY 2.4 (Concavity). *The average footprint $fp(x)$ is a concave function.*

Since $\frac{fp(l+x)-fp(l)}{x} \approx \sum_{t=l+1}^{\infty} P(rt = t)$, $fp(l)$ always increases but increases at a slower rate for a larger l. The function is obviously concave. In the higher order relation, the concavity guarantees that the miss ratio predicted by HOTL is non-increasing with the cache size (as expected from the inclusion property [31]).

2.8 Comparison with Working Set Theory

The first higher-order locality theory is the working set theory, pioneered in Peter Denning's thesis work [13]. His 1968 paper established the relation between the working set size, the miss rate, and the inter-reference interval (iri). The last one is the same as reuse time. The notion of reuse distance or the LRU stack distance was not formalized until 1970 [31]. Figure 4 shows the parallels between the working set locality theory (WSLT) and the new cache locality theory of this paper (CLT).

HOTL hierarchy	working set locality theory (WSLT)	cache locality theory (CLT)
data volume (3rd order)	mean WS size s(T)	mean footprint fp(T), mean fill time vt(c)
miss rate (2nd order)	time-window miss rate m(T), lifetime L(T)=1/m(T)	LRU miss rate mr(c), inter-miss time im(c)=1/mr(c)
reference behavior (1st order)	inter-reference interval (reuse time) distribution P(iri=x)	reuse distance distribution P(rd=x)

Figure 4: Comparison between two higher order locality theories: the working set locality theory (WSLT) for dynamic partitioned primary memory and the cache locality theory (CLT) for cache memory.

WSLT computes the metrics bottom-up. The base metric, $P(iri = x)$, is the histogram of the inter-reference intervals (reuse time), measured in linear time in a single pass of the address trace. The time-window miss ratio $m(T)$ is the sum of reuse time. The mean working set size $s(T)$ is the sum of $m(T)$.

$$m(T) = P(rt > T)$$
$$s(T+1) = s(T) + m(T)$$

Taking together, the working set size $s(T)$ is the second order sum of the reuse frequency.

The $s(T)$ formula was first proved by Denning and Schwartz in 1972 [15]. The formulation assumes an infinitely long execution with a "stationary" state ("the stochastic mechanism ... is stationary"). The working set, $w(t, T)$, is the number of distinct pages accessed between time $t - T + 1$ and t. The average working set size, $s(T)$, is the limit value when taking the average of $w(t, T)$ for all t. The proof is based on the fact that only recurrent pages with an infinite number of accesses contribute to the mean working set size.

In 1978, Denning and Slutz defined the generalized working set (GWS) as a time-space product [16]. The product, denoted here as $st(T)$, is defined for finite-length execution traces, variable-size memory segments, all cache replacement policies that observe the stack property. Interestingly, they found the same recursive relation. The GWS formula is as follows, where the last term is the extra correction to take into account the finite trace length.

$$st(T+1) = st(T) + Tm(T) - a(T)$$

Dividing both sides by T, we have the last term vanishing for large T and see the same recursive relation for GWS in finite-length traces as $s(T)$ in infinitely long traces.

In the present paper, the same recurrence emerges in Section 2.7 as an outcome of Theorem 2.2. For the average footprint, we have effectively

$$fp(T+1) = fp(T) + m(T)$$

If we view the following three as different definitions of the working set: the limit value in 1972 [15], the time-space product in 1978 [16], and the average footprint in 2011 [46], we see an identical equation which Denning envisioned more than four decades ago (before the first proof in 1972). We state it as a law of locality and name it after its inventor:

Denning's Law of Locality *The working set is the second-order sum of the reuse frequency, and conversely, the reuse frequency is the second-order difference of the working set.*

As the relativity theory gives the relation between space and time, Denning's law gives the relation between memory and computation: the working set is the working memory, and the reuse frequency is a summary of program actions (time transformed into frequency and a spectrogram of time). The law states that the relation is higher order.

Our work augments Denning's law in two ways. First, it is the final step to conclusively prove Denning's Law — that it holds for the footprint working set in finite-length program executions. The 1972 proof depends on the idealized condition in infinite-length executions. Subsequent research has shown that the working set theory is accurate and effective in managing physical memory for real applications [14]. The new proof subsumes the infinitely long case and makes Denning's law a logical conclusion for all (long enough) executions. It gives a theoretical explanation to the long observed effectiveness of the working set theory in practice.

Second, we extend HOTL to include cache memory. For main memory, the locality is parameterized in time: the working set of a program in a time quantum. For cache, the primary constraint is space: the miss ratio for a given cache size. Denning et al. named them the "time-window miss ratio" and the "LRU miss ratio" and noted that the two are not necessarily equal [15, 16]. The following formulas show the two miss ratios:

working set	$m(T) = P(rt > T)$
cache locality	$mr(fp(T)) = P(rt > T)$

In the above juxtaposition, the only difference is the parameter to the miss rate function. In $m(T)$, the parameter is the time window length. In $mr(fp(T))$, the parameter is the cache size. Through the second formula, this work connects the cache size and the reuse frequency. In Section 2.4, we show how the time-centric and the space-centric views have different derivations but the same miss ratio. Then in Section 2.7, we give the reuse-window hypothesis as the condition for correctness, which implies the equality between the time-window miss ratio and the LRU miss ratio.

3. Sampling-based Locality Analysis

The footprint can be analyzed through sampling, e.g. by tracing a window of program execution periodically. Sampling has two benefits. First, by reducing the sampling frequency, the cost can be arbitrarily reduced. Second, sampling may better track a program that has significant phase behavior.

Uniform sampling We implement footprint sampling using a technique pioneered by shadow profiling [32] and SuperPin [42]. When a program starts, we set the system timer to interrupt at some preset interval. The interrupt handler is shown in Figure 5. It forks a sampling task and attaches the binary rewriting tool Pin [29]. The Pin tool instruments the sampling process to collect its data access trace, measures all-window footprints using the Xiang formula [46]. In the meanwhile, the base program runs normally until the next interrupt.

Require: This handler is called whenever a program receives the timer interrupt
1: $pid \leftarrow fork()$
2: **if** $pid = 0$ **then**
3: Attach the Pin tool and begin sampling until seeing c distinct memory accesses
4: Exit
5: **else**
6: Reset the timer to interrupt in k seconds
7: Return
8: **end if**

Figure 5: The timer-interrupt handler for footprint sampling

Footprint Sampling Footprint by definition is amenable to sampling. We can start a sample at any point in an execution and continue until the sample execution accesses enough data to fill the largest cache size of interest. We can sample multiple windows independently, which means they can be parallelized. It does not matter whether the sample windows are disjoint or overlapping, as long as the choice of samples is random and unbiased.

The Associative Cache A program execution produces a series of m samples at regular intervals, $x_1, x_2, \ldots, x_m$. We use them in the following way:

1. For each sample x_i, with trace length n_i, predict the miss ratio function $mr(x_i, c)$ for each cache size c by the following:
   (a) Use the analysis of Xiang et al. [46] to compute the average footprint function fp.
   (b) Use the footprint conversion to compute the capacity miss ratio for cache size c.

(c) Use the miss-ratio conversion to compute the reuse distance distribution and the Smith formula [37] to estimate the number of conflict misses for cache size c.

2. For all x_i, take the weighted average and compute the miss ratio for all cache sizes for the program $mr(c) = \frac{\sum_{i=1}^{m} mr(x_i,c)*n_i}{\sum_{i=1}^{m} n_i}$.

The Phase Effect The preceding design assumes phase behavior. Since different samples may come from different phases, combining their footprints would lose the phase distinction. To validate the conjecture, we will compare the phase-sensitive sampling with phase-insensitive sampling. The former, as just described, computes the miss ratio for each sample and then takes the average. The next design combines the footprint from all the samples and then computes the miss ratio. Specifically, the second design is as follows:

1. For each sample x_i, with trace length n_i,
   - Use the analysis of Xiang et al. [46] to compute the average footprint function fp.
2. For all samples x_i, take the weighted average and compute the fp function for the program $fp = \frac{\sum_{i=1}^{m} fp(x_i)*n_i}{\sum_{i=1}^{m} n_i}$.
3. Use the footprint and miss-ratio conversions and the Smith formula [37] to estimate the number of cache misses.

Comparison with Reuse Distance Sampling To be statistically sound, reuse distance sampling must evenly sample reuse windows. After picking an access, it needs to trace the subsequent program accesses until the next data reuse. When a reuse window is long, it does not know a priori how long to monitor, so it has to keep analyzing until seeing the next reuse or until the reuse distance exceeds the largest cache size of interest. The cut-off strategy is also used in footprint sampling.

Beneath this similarity lies two important differences. The reuse distance measures the locality by examining reuses. The footprint measures the locality by examining data accesses. Footprint sampling computes the distribution of all reuse distances from a single sample window using the HOTL conversion. The footprint analysis and conversion take linear time. In comparison, each reuse window sample produces just one reuse distance. It takes asymptotically higher time cost to measure the reuse distance in the sample (than it takes HOTL conversion to compute all reuse distances from the same sample). Hence the advantage of footprint sampling is algorithmic and computational, and this strength comes from the HOTL theory.

$benchname$	n (10^{11})	m $(10^7 bytes)$	T (sec)
400.perlbench	4.2	24.4	457
401.bzip2	1.7	39.3	263
403.gcc	0.2	40.4	72
410.bwaves	14.4	98.2	1664
416.gamess	4.8	0.3	444
429.mcf	1.2	175.7	1172
433.milc	3.8	74.2	1077
434.zeusmp	5.7	51.9	1555
435.gromacs	9.8	1.4	1272
436.cactusADM	20.6	65.5	3411
437.leslie3d	6.8	12.9	1212
444.namd	6.8	4.7	915
445.gobmk	0.9	2.7	173
447.dealII	7.3	88.5	773
450.soplex	1.0	16.2	604
453.povray	4.8	0.3	493
454.calculix	9.5	16.4	1512
456.hmmer	4.9	4.2	303
458.sjeng	7.0	18.2	1356
459.GemsFDTD	8.6	86.9	1397
462.libquantum	3.0	16.8	1391
464.h264ref	2.6	2.7	143
465.tonto	10.0	5.2	1312
470.lbm	3.3	42.9	1491
471.omnetpp	2.3	17.6	1048
473.astar	1.4	29.5	512
481.wrf	9.7	76.8	1895
482.sphinx3	8.9	5.1	1765
483.xalancbmk	3.6	43.8	778

Table 1: The SPEC2006 integer and floating-point benchmarks. For each benchmark, n is the memory trace length of whole execution, m is the number of distinct data blocks (size in bytes) accessed during the execution, and T is the execution time without any instrumentation or analysis.

4. Evaluation

4.1 Experimental Setup

We have tested the full set of 29 benchmarks from SPEC 2006 and 8 from the PARSEC v2.1 suite. All programs are instrumented by Pin [29] and profiled on a Linux cluster where each node has two Intel Xeon 3.2GHz processors. PARSEC is run on a machine with two Intel Xeon E5649 processors. In simulation, we simulate a single-level cache, which is shared in the case of parallel code. On a real machine, the baseline is the program run time without instrumentation or any analysis.

For SPEC 2006, we use the first reference input provided by the benchmark suite. Table 1 shows for each SPEC 2006 program the length of trace n, the size of data m and the time of the unmodified program execution. The length of SPEC 2006 traces ranges from 20 billion in *403.gcc* to 2.1 trillion in *436.cactusADM*. The amount of data ranges from 3MB in *416.gamess* to 1.7GB in *429.mcf*. For PARSEC, we test programs using the three provided input sizes: simsmall, simmedium and simlarge. We run each with 4 threads, a commonly used configuration.

Locality sampling is implemented using fork, as described in Section 3. The implementation does not yet recognize system calls, so sampling handles only 22 of the 29 sequential programs. Nor does the sampling implementation handle multi-threaded code. We evaluate miss-ratio prediction using the full trace of the 8 parallel programs.

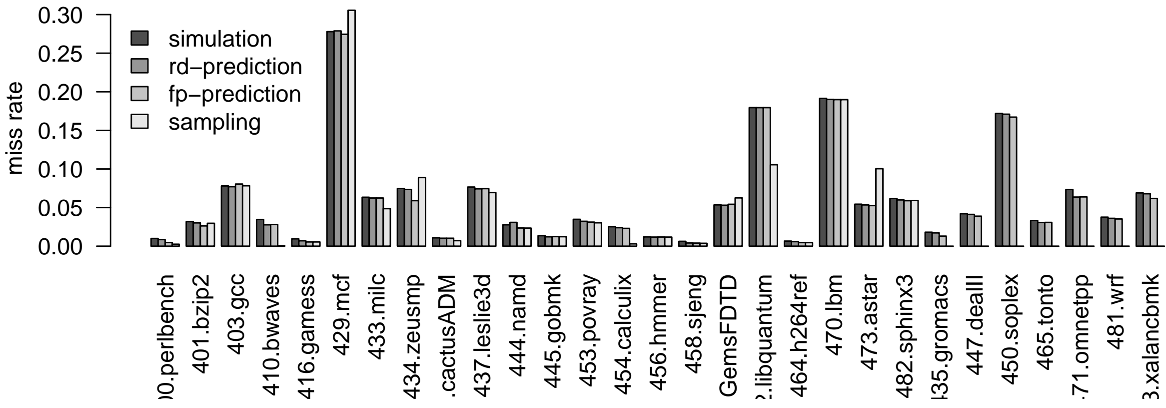
3073 Cache Sizes In the analysis, the footprint and reuse distance numbers are bin-ed using logarithmic ranges as follows. For each (large enough) power-of-two range, we sub-divide it into (up to) 256 equal-size increments. As a result, we can predict the miss ratio not just for power-of-two cache sizes, but 3073 cache sizes between 16KB and 64MB.

4.2 Miss-Ratio Prediction

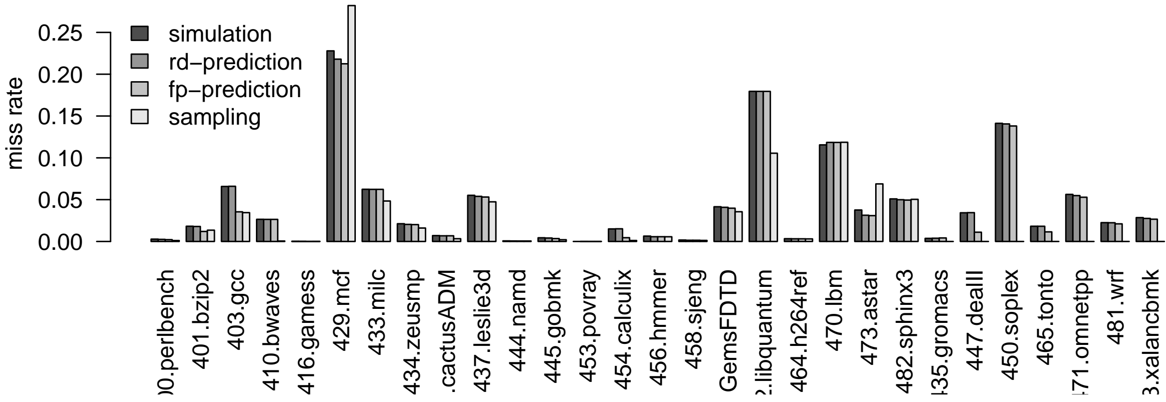
We first evaluate the accuracy and the speed of miss-ratio prediction, made by the Filmer conversion and locality sampling, tested on sequential and parallel programs, and verified through simulation and hardware counters.

4.2.1 Sequential Programs

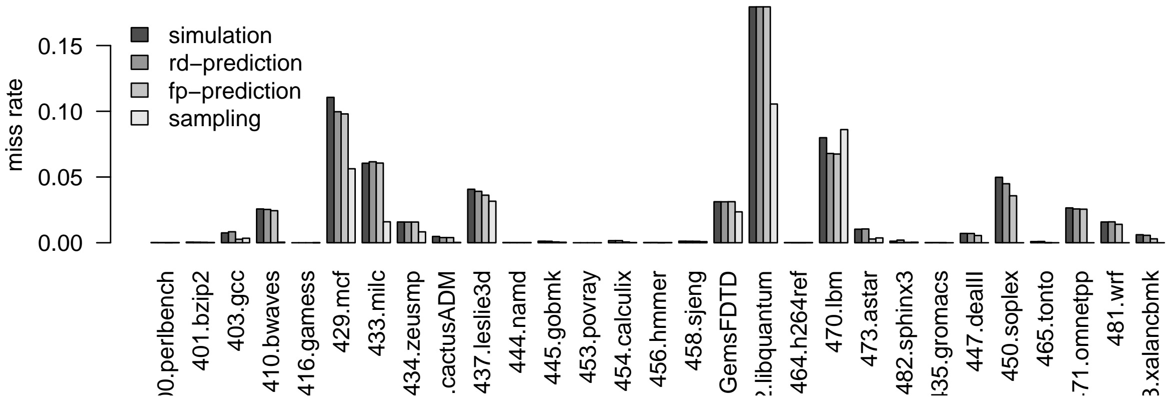
We first use cache simulation to evaluate the accuracy of Filmer-based miss ratio prediction. Instead of evaluating each of the 29



(a) 8-way, 32KB cache



(b) 8-way, 256KB cache



(c) 16-way, 8MB cache

Figure 6: Accuracy of the miss-ratio prediction by reuse distance, footprint (HOTL conversion in Section 2.4), and footprint sampling (Section 3) for 29 SPEC 2006 benchmarks, each on 3 (of the 3073) cache configurations, compared with cache simulation. (a) 8-way, 32KB cache. (b) 8-way, 256KB cache. (c) 16-way, 8MB cache. The sampling results are for 22 out of 29 programs (not the last 7). The average time cost of sampling is 0.5% (Table 2).

programs on 3073 cache sizes, we show results for 3 configurations: 32KB, 8-way associative L1D; 256KB, 8-way associative L2; and 8MB, 16-way associative L3. They are the cache configurations on our test machine. We will compare our prediction to the performance counter result later. The cache-block size is 64 bytes in all cases. The accuracy for the other 3070 cache sizes looks similar.

Figure 6 plots the measured and predicted miss ratios. The *rd-prediction*, which uses the measured reuse distance, is the most accurate. The other two, *fp-prediction* and *sampling* measure the footprint and then use the HOTL conversion in Section 2.4. The HOTL conversion *fp-prediction* closely matches the reuse-distance analysis *rd-prediction* in almost all cases, showing that the footprint-converted reuse distance is almost identical to the measured reuse distance—hence the validity of the reuse-window hypothesis.

The Phase Effect The reuse distances of a program, when added together regardless of phases, predict the (capacity) miss ratio accurately, because an access is a cache capacity miss if and only if its reuse distance is greater than the cache size. On the other hand, the footprint should be affected by phases. As the footprint changes from phase to phase, it is possible that taking the global average might lose critical information.

A consistent result from the theory and the experiments is that the two are largely equivalent, as far as computing the miss ratio is concerned. The theory gives the conversion procedure from the footprint to the reuse distance. The experiments show, by the close match between *rd-prediction* and *fp-prediction* in Figure 6, the conversion is accurate for most programs. This suggests that reuse windows are representative of all windows, and this is why the prediction is accurate in spite of the phase effect.

Another evidence, for which we do not include the results in the paper, is that the two sampling designs in Section 3, phase sensitive and insensitive, produce almost identical predictions.

Analysis Speed Table 2 compares the cost of four analysis methods: the simulation sim, the reuse distance rd [49], the footprint fp [46], and the footprint sampling sp. For simulation we could use the algorithm of Smith and Hill [25] to simulate all three configurations in one pass. For speed comparison, we ran the simplest simulator once for each configuration. The simulation cost in table 2 is the average of the three runs.

The cost for reuse distance analysis ranges from 52 times to 426 times with an average of 153 times. The footprint analysis costs about 7 times less, with slowdowns between 6 and 66 times and on average 23 times. Simulation for a single configuration has slowdowns from 14 to 80 times, with an average of 38 times.

Comparing the average, we see that measuring the footprint, the third order Filmer metric that can compute the second order metric miss ratio for all cache sizes, is 39% faster than simulating for a single cache size, before we use footprint sampling.

4.2.2 Locality Sampling

For this experiment, we choose somewhat arbitrarily the frequency of one sample every 10 seconds. The sample length is the volume fill time for the cache size. Sampling analysis is not always accurate. Visible errors are seen in *mcf*, *libquantum* and *astar* in Figure 6. The reason, as shown by the last column of Table 2, is that it covers less than 1% of the execution. The coverage is computed by the ratio of the number of sampled instructions to the total number of instructions (counted by our full trace profiling). The coverage is as low as 0.006% in *lbm*. The low coverage does not mean low accuracy. The prediction of *lbm* is 99% accurate for the 32KB cache, 97% for the 256KB cache, and 92% for the 8MB cache.

In Table 2, we show the slowdown in the end-to-end run time by the column marked $samp$. It ranges from 0% to 2.14%. Three programs have a visible cost of over 1%. They are *bwaves* 2.1%, *GemsFDTD* 1.3% and *milc* 1.5%. The reason for the relatively high cost may be the non-trivial interference between the sampling task and the parent task. Across all programs, the average visible overhead is below a half percent. If we measure the total CPU time, sampling takes between 0% and 80% of the original run time. The average cost is 19%, of which over 18% is hidden by shadow profiling.

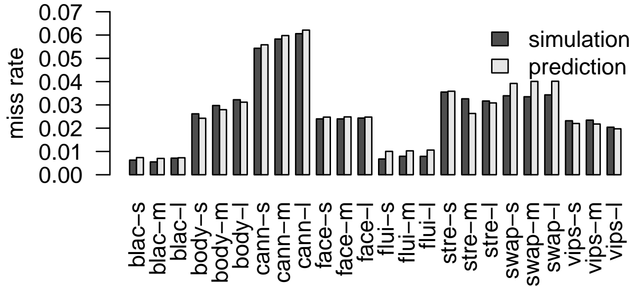
$benchname$	sim	rd	fp	$samp$	cov
400.perlbench	49	219	34	0.24%	3.1%
401.bzip2	34	139	24	0.73%	1.5%
403.gcc	24	88	15	0.55%	0.1%
410.bwaves	57	196	35	2.14%	0.5%
416.gamess	62	286	40	0.29%	2.9%
429.mcf	10	56	6	0.14%	0.03%
433.milc	21	74	9	1.53%	0.04%
434.zeusmp	25	102	14	0.81%	0.06%
435.gromacs	40	142	19	-	-
436.cactusADM	40	167	21	0.00%	1.1%
437.leslie3d	42	131	23	0.00%	0.01%
444.namd	44	155	24	0.00%	2.2%
445.gobmk	35	130	23	0.22%	1.1%
447.dealII	54	209	34	-	-
450.soplex	13	52	7	-	-
453.povray	51	220	33	0.00%	1.8%
454.calculix	39	127	19	0.11%	1.8%
456.hmmer	80	426	59	0.00%	0.8%
458.sjeng	34	152	23	0.82%	0.4%
459.GemsFDTD	42	181	21	1.28%	0.01%
462.libquantum	17	48	9	0.00%	0.01%
464.h264ref	101	424	66	0.00%	1.2%
465.tonto	52	168	30	-	-
470.lbm	14	76	6	0.00%	0.01%
471.omnetpp	17	69	10	-	-
473.astar	15	73	11	0.80%	0.9%
481.wrf	33	113	19	-	-
482.sphinx3	30	117	16	0.59%	1.2%
483.xalancbmk	31	99	21	-	-
average	38	153	23	0.47%	0.9%

Table 2: Time comparison between different profiling methods and cache simulation for SPEC 2006. The baseline is the execution time without any instrumentation or analysis. The middle four columns show the slowdown compared to the baseline: sim for simulating one cache size, rd for reuse distance profiling, fp for footprint profiling, and $samp$ for footprint sampling. The last column cov gives the sampling coverage.

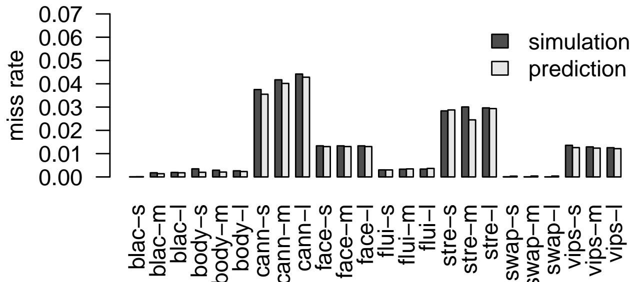
4.2.3 Parallel Programs

Figure 7 shows that for 3 of the 3073 cache configurations and across the 3 input sizes, the predicted miss ratio matches closely with the simulated miss ratio, similar to the results we saw in the sequential programs. The accuracy shows that the reuse-window hypothesis holds for these threaded applications.

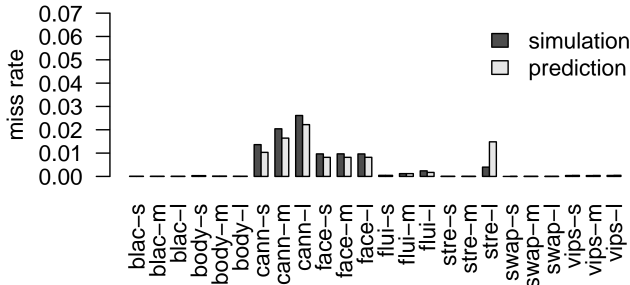
The last column of table 3 shows the slowdowns of footprint profiling, which ranges from 14 times to 159 times with an average of 113 times. We did not profile reuse distance for PARSEC because it took too long. We note that the footprint analysis shows 5 times as much overhead in 4-threaded tests as in sequential programs (159 times in PARSEC vs. 23 times in SPEC 2006). The reason is that our data analysis is still serial, so the overhead is proportional to the total amount of work. We plan to parallelize the



(a) 8-way, 32KB cache



(b) 8-way, 256KB cache



(c) 16-way, 8MB cache

Figure 7: Accuracy of the miss-ratio prediction for 3 (of the 3073) cache configurations and 3 input sizes, compared with cache simulation. (a) 8-way, 32KB cache. (b) 8-way, 256KB cache. (c) 16-way, 8MB cache.

footprint analysis in the future, building on recent work in parallelizing the reuse-distance analysis [12, 22, 33].

4.3 Validation on a Real Machine

In Figure 8, we compare the simulation result with the miss ratio measured by the hardware counters on our test machine. To measure the actual misses, we use Intel's VTune tool to record three hardware counter events named

bench name	*input size*	n (10^9)	m $(10^6 bytes)$	T (sec)	slow-down
black -scholes	S	0.1	0.4	0.093	129
	M	0.4	1.2	0.384	91
	L	1.6	4.4	1.542	88
body -track	S	0.3	8.1	0.285	129
	M	1.1	11.1	0.948	155
	L	4.0	14.8	3.35	111
canneal	S	0.6	43.0	1.525	19
	M	1.3	84.3	3.859	15
	L	2.7	164.9	8.804	14
facesim	S	12.7	344.2	7.448	139
	M	12.7	344.2	7.306	131
	L	12.7	344.2	7.86	116
fluid -animate	S	0.5	10.5	0.429	114
	M	1.3	20.6	0.983	145
	L	3.9	57.6	2.9	124
stream -cluster	S	0.5	1.2	0.722	87
	M	2.6	2.9	1.641	138
	L	9.6	9.5	6.951	173
swapt -ions	S	3.6	0.9	2.349	134
	M	1.4	1.2	0.935	114
	L	5.7	1.9	3.766	132
vips	S	1.0	13.5	0.748	159
	M	3.1	26.9	2.228	140
	L	8.6	15.7	7.332	103

Table 3: The PARSEC parallel benchmarks. For each benchmark, n is the memory trace length of whole execution, m is the size of program data (in bytes) accessed during the execution, and T is the execution time without any instrumentation or analysis. The last column is the slowdown of the footprint analysis.

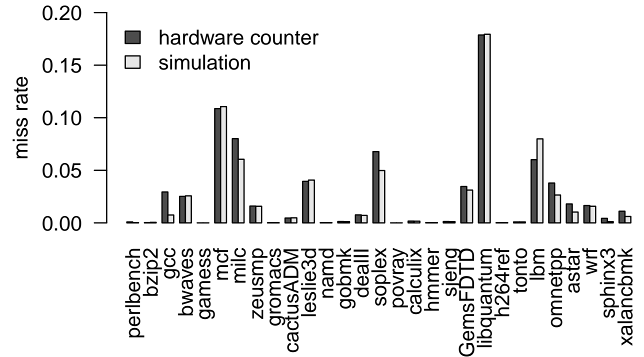


Figure 8: Comparison between hardware counter measured L3 cache miss ratio and the simulation result.

```
OFFCORE_RESPONSE_0.DATA_IN.LOCAL_DRAM
MEM_INST_RETIRED.LOADS
MEM_INST_RETIRED.STORES
```

The measured miss ratio is the first count divided by the sum of the last two counts.

The figure shows a significant difference in *gcc*. The reason is that the simulation considers only data accesses but the hardware counter counts instruction misses in the data cache, which we believe are significant in *gcc*.

4.4 Direct Fill Time vs. Filmer Fill Time

The measurement of the direct fill time, definition in Section 2.3 and algorithm in Section A, takes so long that the only programs we could finish are 10 of the 11 SPEC 2000 integer benchmark programs. Table 4 compares the average time for these programs. An unmodified SPEC 2000 program runs for 3 minutes on average, the direct fill time analysis takes over 22 hours. The average overhead is more than 7 hours for each minute. In comparison, the per minute overhead is an hour and a half for reuse distance and 7 minutes if we first compute footprint and then derive the Filmer fill time.

analysis	avg. time	avg. slowdown
direct fill time (Section A)	22h12m11s	446x
reuse distance	3h57m36s	84x
Filmer fill time (Section 2.3)	22m4s	8x

Table 4: Speed comparison for 10 SPEC 2000 integer benchmarks. The average trace length n is 47 billion, data size m is 73MB, and baseline run time is 3 minutes and 16 seconds.

More problematic is that with the direct fill time, the predicted miss ratio is not monotone. Worse, the miss ratio may be negative. Consider an example trace with 100 a's followed by 11 b's, 1 c, 20 d's, 15 e's, 1 f and 320 g's. The average time to fill a 4-element cache, $vt(4)$, is 161.5, is longer than the average time to fill a 5-element cache, $vt(5)$, which is 149.5. Since the direct fill time decreases when the cache size v increases, the predicted miss ratio is negative!

The preceding example was constructed based on an analysis of real traces. During experimentation, we found that the miss ratios of some cache sizes were negative. While most of the 3000 or so sizes had positive predictions, the negatives were fairly frequent and happened in most test programs. It seemed contradictory that it could take a program longer to fill a smaller cache. The reason is subtle. To compute the direct fill time, we find windows with the same footprint and take the average length. As we increase the footprint by 1, the length of these windows will increase but the number of such windows may increase more, leading to a lower average, as happened in the preceding example.

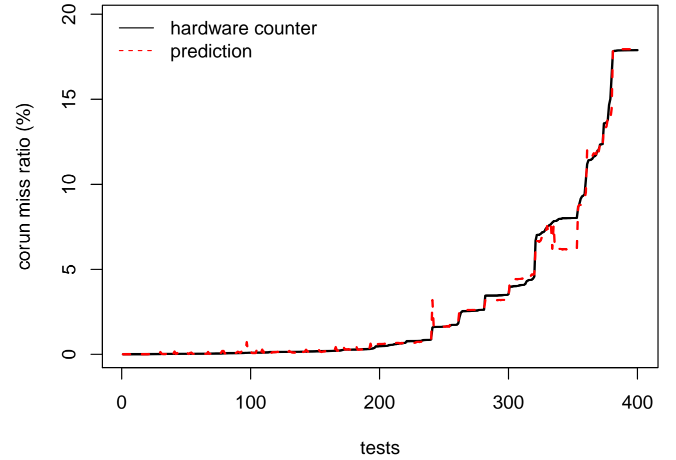
In contrast, the Filmer fill time is a positive, concave function (Corollary 2.4). Its miss-ratio prediction is monotone and can be measured in near real time (Section 4.2.2).

4.5 Predicting Cache Interference

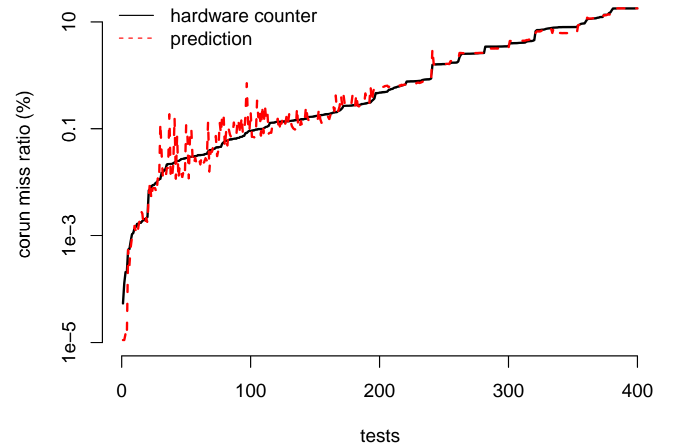
A complete 2-program co-run test for the 29 SPEC 2006 benchmarks would include $\binom{29}{2} = 406$ program pairs. To reduce the clutter in the graphs we show, we choose 20 programs. To avoid bias, we pick programs with the smallest benchmark ids. Since we profile data accesses only, we exclude perlbench and gcc because their large code size may cause significant instruction misses in the data cache. After the removal, we have 20 SPEC benchmark programs from 401.bzip2 to 464.h264ref. The trimming reduces the number of pair-run tests to $\binom{20}{2} = 190$.

Cache interference models were pioneered by Thiebaut and Stone [41], Suh et al. [39] and Chandra et al. [9], who computed the cache interference by the impact of the peer footprint on the self locality.[2] The footprint is measured for a single window length [41] and approximated for multiple lengths [9, 39]. Our subsequent work found a way to measure all-window footprints precisely and efficiently [17, 45, 46]. The self locality is measured by the reuse distance. As the measurement problem for the footprint is solved, the speed of reuse-distance analysis becomes the bottleneck. We found that by profiling up to two days for each program, the reuse distance analyzer by Zhong et al. [49] could finish only 8 SPEC 2006 programs [46]. The total modeling time was over 106 CPU hours, 94% of which was spent on the reuse-distance analysis. In

[2] Chandra et al. also gave a model that used only the reuse distance [9]. Zhuravlev et al. used it and two other such models and found that in task scheduling, they did not significantly outperform a simple model that used only the miss rate [51].



(a) linear scale miss ratios



(b) logarithmic scale miss ratios

Figure 9: The predicted and measured miss ratios of the 380 executions in 190 pair runs. The executions are ordered by the ascending miss ratio as measured by the hardware counters in exhaustive testing. For each execution, the solid (black) line shows the hardware counter result, and the dotted (red) line shows the prediction. The prediction takes about a *half* percent of the time of exhaustive testing. Just two executions have a significant error in both graphs, which are a *half* percent of all executions.

this study, we have measured reuse distance for all benchmarks (see Table 2 for measurement costs). Some programs took over 4 days.

Based on the new theory, we compute the reuse distance from the footprint and predict the co-run interference. Figure 9 compares the measured and predicted miss ratios. There are 190 pair runs for a total of 380 executions. The x-axis orders these executions by the measured miss ratios from the lowest to the highest. For easy viewing, we connect the points into a line. The measured curve is necessarily monotone. The prediction is to match the measurement.

Figure 9 has two graphs, showing the miss ratio in the linear scale in the upper graph and the logarithmic scale in the lower graph. The prediction is mostly accurate. The errors happen but for different executions in the two graphs. If an error is visible in the linear scale but not in the logarithmic scale, the error is significant in absolute terms but not in relative terms. Similarly, we have errors significant relatively but not absolutely. The two graphs show just two errors that are significant in both scales. In the other 378 (99.5%) executions, the prediction is either accurate or the error insignificant. From visual inspection, the error is significant in just 0.5% of all executions.

To make the prediction, the analysis needs 1 hour 4 minutes CPU time for sampling, almost as fast as we can run the 20 programs without analysis. In comparison, the exhaustive testing takes over 9 days (estimated) of CPU time. The cost saving is 99.5%.

To see interference in 3-program co-runs, the exhaustive testing has to re-test and collect results anew, but the modeling needs no additional testing. Indeed, the new model has been used in an on-line system to regroup eight programs to run on two quad-core processors (to have a higher performance or at least a more repeatable performance) [47]. The exhaustive testing of the 4-program co-runs in our 20-program suite would need 19 thousand test executions and have taken months of time.

To summarize the pair interference experiment, we can say that the result is half and half: the modeling takes half percent of the time and has a significant error in a half percent of executions.

5. Related Work

The concept of locality has evolved from an observation that a program does not use all the data at all times, to quantitative metrics that we can evaluate and compare but for which we must solve the dual problems of speed and precision.

Locality sampling A publicly available system for locality sampling is the SLO tool developed by Beyls and D'Hollander [5]. SLO instruments a program to skip every k accesses and take the next address as a sample. A bounded number of samples are kept in a sample reservoir. To track reuse windows, it checks each access to see if it is an access to some sampled datum. The instrumentation code is carefully engineered in GCC to have just two conditional statements for each memory access (one for address and the other for counter checking). Reservoir sampling reduces the time overhead from 1000-fold slow-down to only a factor of 5 and the space overhead to within 250MB extra memory. The sampling accuracy is 90% with 95% confidence. The accuracy is measured in the reuse time, not the reuse distance or the miss ratio.

To accurately measure reuse distance, a record must be kept to count the number of distinct data appeared in a reuse window. Zhong and Chang developed the bursty reuse distance sampling, which divides a program execution into sampling and hibernation periods [48]. In the sampling period, the counting uses a tree structure and costs $O(\log \log M)$ per access. If a reuse window extends beyond a sampling period into the subsequent hibernation period, the counting uses a hash-table, which reduces the cost to $O(1)$ per access. Multicore reuse distance analysis uses a similar scheme for analyzing multi-threaded code [35]. Its fast mode improves over hibernation by omitting the hash-table access at times when no samples are being tracked. Both methods compute the reuse distance accurately.

StatCache is based on unbiased uniform sampling [3]. After a data sample is selected, StatCache puts the page under the OS protection to capture the next access to the same datum. It uses the hardware counters to measure the time distance till the reuse. OS protection is limited by the page granularity. Two other systems, developed by Cascaval et al. [7] and Tam et al. [40], used the special support on IBM processors to trap accesses to specified data addresses. To reduce the cost, these methods used a small number of samples. Cascaval et al. used the Hellinger Affinity Kernel to infer the accuracy of sampling [7]. Tam et al. predicted the miss rate curves in real time [40].

Locality measurement Reuse distance is a shorter name for the LRU stack distance defined by Mattson et al. [31]. The fastest precise method takes $O(n \log m)$ time, where n is the length of the trace and m is the size of data [34]. A variation of the algorithm powered the Cheetah cache simulator [38], widely distributed as part of the SimpleScalar tool set. By approximating long-distance reuses (with a guaranteed precision e.g. 99%), the cost can be reduced to $O(n \log \log m)$ [49]. This $n \log \log m$ algorithm is used in the two most recent sampling studies [35, 48]. In our experiments, the cost is several hundred times slowdown. The average cost reported in another study is as high as several thousand times slowdown (although with a different implementation) [35]. Zhong et al. gave a lower bound result indicating that the (asymptotic) cost cannot be further reduced for full reuse distance analysis [49]. Recent studies found efficient algorithms to parallelize the reuse distance analysis to run on MPI [33] or GPU [12, 22].

Time-based conversion [27, 36] and StatStack [19, 20] each gave a statistical formula to convert the reuse time distribution to miss rate, so did the working set theory [15]. These methods were not guaranteed to be correct or have a bounded error. This work gives a different conversion method based on the footprint formula and the correctness condition for the conversion.

If the cost of measuring $O(n)$ reuse windows was high, the cost of measuring $O(n^2)$ footprint windows was prohibitively high. In 2008, a sub-quadratic cost $O(n \log m)$ solution was proposed [17]. Later, the algorithm was implemented and made 70 times faster [45]. These two methods measure the full distribution, including for example, the maximum and the minimum sizes. Instead of the full distribution, Xiang et al. showed that the average footprint can be measured in linear time $O(n)$, and it is a monotone function [46]. Based on the HOTL theory in this paper, we have reduced the analysis cost to a negligible level using sampling and proved that the footprint function is concave.

Program sampling Arnold and Ryder pioneered a general framework to sample Java code, i.e. the first few invocations of a function or the beginning iterations of a loop [2]. It has been adopted for hot-stream prefetching in C/C++ in bursty sampling [11] and extended to sample both static and dynamic bursts for calling context profiling [50]. Shadow profiling pauses a program at preset intervals and forks a separate process to profile in parallel with the base program [32, 42]. Before the new theory, the reuse distance analysis is not a good target for these techniques because of the uncertain length of the reuse windows. With the new theory, locality sampling becomes a similar task as frequency profiling. Like frequency profiling, the cost can be adjusted by simply changing the sampling rate.

Filmer metrics in multi-threaded code The locality metrics in particular the footprint and the reuse distance have been extended to multi-threaded code by a number of studies, including composable modeling of shared footprint [18], statistical modeling in con-

current reuse distance [27], and direct measurement by multi-core reuse distance [35]. In a concurrent program, the reuse distance is affected by data sharing, thread interleaving and composition. These studies solved the problems by characterizing the relation between the private reuse distance (PRD) and the concurrent reuse distance (CRD). For loop-based code, Wu and Yeung gave a scaling model to predict how the reuse distance changes when the work is divided by a different number of threads [43]. These modeling techniques have found uses in co-scheduling [26] and multicore cache hierarchy design [44]. In this paper, we use footprint sampling and HOTL conversion in multi-threaded code and show the result that the reuse-window hypothesis holds there as it does in sequential code.

6. Summary

In this paper, we have compiled five Filmer metrics— the footprint, the inter-miss time, the volume fill time, the miss ratio curve and the reuse distance—and shown that they are mutually derivable. The derivations form a higher order relation. We prove that two of the miss-ratio derivations, by the footprint and by the reuse time, are mathematically equivalent. As a result, the correctness of the conversion depends on the reuse-window hypothesis. In addition, we prove that the average footprint is a concave function. We also give a direct definition of the fill time and show it to be unusable in practice. When comparing with the working set theory, we show the recurring theoretical result which we call Denning's law of locality. We show how the new theory complements and extends the previous theory.

Based on the new theory, we have developed a novel technique of locality sampling and used it to predict the miss ratio. When tested on the full suite of the SPEC 2006 benchmarks, the HOTL conversion predicts the miss ratio for over 3000 cache sizes at a speed 39% faster than cache simulation for a single cache size. The prediction is accurate compared to simulation and hardware counter results. Locality sampling obtains a similar accuracy by examining 0.9% of the execution and incurring a cost of less than 0.5% of the time of the unmodified code. When used to predict cache interference, the new technique takes 0.5% of the time of the exhaustive testing and predicts the interference accurately for 99.5% of the executions.

In summary, we have shown that the Filmer metrics can be measured in real time, and they are easy to compose and convert. We expect that the higher order theory and the sample technique will provide a new foundation for developing future techniques of locality analysis and optimization.

Acknowledgments

The comparison with the working set theory was done in collaboration with Peter Denning. It was a rare privilege to discuss the field defining ideas with their creator. He was also the first to use the acronyms HOTL, WSLT and CLT when commenting on our paper and suggested the comparative view in Figure 4. Kim Hazelwood, Ramesh Peri and Tipp Moseley answered our questions about Pin and shadow profiling. We also thank Jacob Brock, Xipeng Shen, Donald Yeung, other colleagues, the reviewers of ASPLOS and the program committee especially P. Sadayappan for the careful review and constructive critiques, which are invaluable in improving the presentation of both the theory and the evaluation.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, Oct. 2001.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of PLDI*, pages 168–179, Snowbird, Utah, June 2001.

[3] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Proceedings of SIGMETRICS*, pages 169–180, 2005.

[4] K. Beyls and E. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.

[5] K. Beyls and E. D'Hollander. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of HPCC. Springer. Lecture Notes in Computer Science Vol. 4208*, pages 220–229, 2006.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of PACT*, pages 72–81, 2008.

[7] C. Cascaval, E. Duesterwald, P. F. Sweeney, and R. W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of PACT*, pages 339–349, 2005.

[8] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of ICS*, pages 150–159, 2003.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of HPCA*, pages 340–351, 2005.

[10] A. Chauhan and C.-Y. Shei. Static reuse distances for locality-based optimizations in MATLAB. In *Proceedings of ICS*, pages 295–304, 2010.

[11] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of PLDI*, Berlin, Germany, June 2002.

[12] H. Cui, Q. Yi, J. Xue, L. Wang, Y. Yang, and X. Feng. A highly parallel reuse distance analysis algorithm on gpus. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2012.

[13] P. J. Denning. The working set model for program behaviour. *Commun. ACM*, 11(5):323–333, 1968.

[14] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), Jan. 1980.

[15] P. J. Denning and S. C. Schwartz. Properties of the working set model. *Communications of ACM*, 15(3):191–198, 1972.

[16] P. J. Denning and D. R. Slutz. Generalized working sets for segment reference strings. *Communications of ACM*, 21(9):750–759, 1978.

[17] C. Ding and T. Chilimbi. All-window profiling of concurrent executions. In *Proceedings of PPoPP*, 2008. *poster paper*.

[18] C. Ding and T. Chilimbi. A composable model for analyzing locality of multi-threaded programs. Technical Report MSR-TR-2009-107, Microsoft Research, August 2009.

[19] D. Eklov, D. Black-Schaffer, and E. Hagersten. Fast modeling of shared caches in multicore systems. In *Proceedings of HiPEAC*, pages 147–157, 2011. *best paper*.

[20] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Proceedings of ISPASS*, pages 55–65, 2010.

[21] C. Fang, S. Carr, S. Önder, and Z. Wang. Path-based reuse distance analysis. In *Proceedings of CC*, pages 32–46, 2006.

[22] S. Gupta, P. Xiang, Y. Yang, and H. Zhou. Locality principle revisited: A probability-based quantitative approach. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2012.

[23] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[24] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, University of California, Berkeley, Nov. 1987.

[25] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[26] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of HiPEAC*, pages 201–215, 2010.

[27] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Proceedings of CC*, pages 264–282, 2010.

[28] S. F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Flexible reference trace reduction for VM simulations. *ACM Transactions on Modeling and Computer Simulation*, 13(1):1–38, 2003.

[29] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of PLDI*, pages 190–200, 2005.

[30] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of SIGMETRICS*, pages 2–13, 2004.

[31] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[32] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of CGO*, pages 198–208, 2007.

[33] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan. PARDA: A fast parallel reuse distance analysis algorithm. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2012.

[34] F. Olken. Efficient methods for calculating the success function of fixed space replacement policies. Technical Report LBL-12370, Lawrence Berkeley Laboratory, 1981.

[35] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of PACT*, pages 53–64, 2010.

[36] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of POPL*, pages 55–61, 2007.

[37] A. J. Smith. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of ICSE*, 1976.

[38] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of SIGMETRICS*, Santa Clara, CA, May 1993.

[39] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of ICS*, pages 1–12, 2001.

[40] D. K. Tam, R. Azimi, L. Soares, and M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of ASPLOS*, pages 121–132, 2009.

[41] D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, 1987.

[42] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of CGO*, pages 209–220, 2007.

[43] M.-J. Wu and D. Yeung. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of PACT*, pages 264–275, 2011.

[44] M.-J. Wu and D. Yeung. Identifying optimal multicore cache hierarchies for loop-based parallel programs via reuse distance analysis. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance and Correctness*, pages 2–11, 2012.

[45] X. Xiang, B. Bao, T. Bai, C. Ding, and T. M. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of PPoPP*, pages 91–102, 2011.

[46] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Proceedings of PACT*, pages 350–360, 2011.

[47] X. Xiang, B. Bao, C. Ding, and K. Shen. Cache conscious task regrouping on multicore processors. In *Proceedings of CCGrid*, pages 603–611, 2012.

[48] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proceedings of ISMM*, pages 91–100, 2008.

[49] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM TOPLAS*, 31(6):1–39, Aug. 2009.

[50] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of PLDI*, pages 263–271, 2006.

[51] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of ASPLOS*, pages 129–142, 2010.

A. Measuring the Direct Fill Time

As defined in Section 2.3, the direct fill time is the average length of all windows that have the same-size footprint. For a trace of n accesses to m data, the fill time algorithm counts all $O(n^2)$ windows but reduces the quadratic cost of counting in three ways. The solution is similar in design to all-window footprint measurement [17, 45].

Counting by footprint size rather than window length The footprint size in a window is up to m, the size of data. Although there are up to n windows ending at each element, there are at most m different footprint sizes. By counting m footprint sizes rather than n windows, the algorithm reduces the counting cost from $O(n^2)$ to $O(nm)$.

Consider the example in Figure 10. Take the trace till the second access of b (before |). It is the 6th access, so there are 6 windows ending there. Only 3 distinct elements are accessed, so the 6 windows have at most 3 different footprint sizes. From small to large, the 6 windows have a length 1 to 6 and footprints 1,2,3,3,3,3 respectively.

$aabacb \,|\, acadaadeedab$

Windows ending at the second b

b, cb, acb, bacb, abacb, aabacb

Figure 10: There are 3 different footprints for the 6 windows ending at the second b, so the 6 windows can be counted in 3 (instead of 6) steps.

Relative precision footprint size By measuring data sizes with a relative precision, for example, 99% or 99.9%, the number of different footprint sizes becomes $O(\log m)$ instead of m. The cost of the algorithm becomes $O(n \log m)$.

Trace compression A user sets a positive threshold c. The trace is divided into a series of k intervals. Each interval has c distinct elements (except for the last interval, which may have fewer than c distinct elements). This is known as trace compression [28]. The algorithm traverses the trace interval by interval rather than element by element. The length of the trace is reduced from n to k, and the cost becomes $O(ck \log m)$.

The algorithm computes the full distribution of the fill time $VT(v)$, from which we can compute the average fill time $vt(v)$. As far as we know, this is the first algorithm that computes the direct fill time with a guaranteed precision. We have implemented it and shown the results in the evaluation section.

Program Locality Analysis Using Reuse Distance

YUTAO ZHONG
George Mason University
XIPENG SHEN
The College of William and Mary
and
CHEN DING
University of Rochester

On modern computer systems, the memory performance of an application depends on its locality. For a single execution, locality-correlated measures like average miss rate or working-set size have long been analyzed using *reuse distance*—the number of distinct locations accessed between consecutive accesses to a given location. This article addresses the analysis problem at the program level, where the size of data and the locality of execution may change significantly depending on the input.

The article presents two techniques that predict how the locality of a program changes with its input. The first is approximate reuse-distance measurement, which is asymptotically faster than exact methods while providing a guaranteed precision. The second is statistical prediction of locality in all executions of a program based on the analysis of a few executions. The prediction process has three steps: dividing data accesses into groups, finding the access patterns in each group, and building parameterized models. The resulting prediction may be used on-line with the help of distance-based sampling. When evaluated on fifteen benchmark applications, the new techniques predicted program locality with good accuracy, even for test executions that are orders of magnitude larger than the training executions.

The two techniques are among the first to enable quantitative analysis of whole-program locality in general sequential code. These findings form the basis for a unified understanding of program

The article contains material previously published in the 2002 Workshop on Languages, Compilers, and Runtime Systems (LCR), 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), and 2003 Annual Symposium of Los Alamos Computer Science Institute (LACSI).
The authors were supported by the National Science Foundation (CAREER Award CCR-0238176 and two grants CNS-0720796 and CNS-0509270), the Department of Energy (Young Investigator Award DE-FG02-02ER25525), IBM CAS Faculty Fellowship, and a gift from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.
Authors' addresses: Y. Zhong, George Mason University, Fairfax, VA; email: yzhong@cs.gmu.edu; X. Shen, College of William and Mary, Williamsburg, VA; email: xshen@cs.wm.edu; C. Ding, University of Rochester, Rochester, NY; email: cding@cs.rochester.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 0164-0925/2009/08-ART20 $10.00
DOI 10.1145/1552309.1552310 http://doi.acm.org/10.1145/1552309.1552310

locality and its many facets. Concluding sections of the article present a taxonomy of related literature along five dimensions of locality and discuss the role of reuse distance in performance modeling, program optimization, cache and virtual memory management, and network traffic analysis.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization, compilers*

General Terms: Measurement, Languages, Algorithms

Additional Key Words and Phrases: Program locality, reuse distance, stack distance, training-based analysis

ACM Reference Format:

Zhong, Y., Shen, X., and Ding, C. 2009. Program locality analysis using reuse distance. ACM Trans. Program. Lang. Syst. 31, 6, Article 20 (August 2009), 39 pages.
DOI = 10.1145/1552309.1552310 http://doi.acm.org/10.1145/1552309.1552310

1. INTRODUCTION

Today's computer systems must manage a vast amount of memory to meet the data requirements of modern applications. Because of fundamental physical limits—transistors cannot be infinitely small and signals cannot travel faster than the speed of light—practically all memory systems are organized as a hierarchy with multiple layers of fast cache memory. On the software side, the notion of *locality* arises from the observation that a program uses only part of its data at each moment of execution. A program can be said to conform to the 80-20 rule if 80% of its execution requires only 20% of its data. In the general case, we need to measure the active data usage of a program to understand and improve its use of cache memory.

Whole-program locality describes how well the data demand of a program can be satisfied by data caching. Although a basic question in program understanding, it has eluded systematic analysis in the past due to two main obstacles: the complexity of program code and the effect of program input. In this article, we address these two difficulties using training-based locality analysis. This analysis examines the execution of a program rather than analyzing its code. It profiles a few runs of the program and uses the result to build a statistical model to predict how the locality changes in other runs. Conceptually, training-based analysis is analogous to observation and prediction in the physical and biological sciences.

The basic runtime metric we measure is *reuse distance*. For each data access in a sequential execution, the reuse distance is the number of *distinct* data elements accessed between the current and previous accesses to the same datum (the distance is infinite if no prior access exists). It is the same as the LRU stack distance defined by Mattson et al. [1970]. As an illustration, Figure 1(a) shows an example access trace and its reuse distances. If we take the histogram of all (finite) reuse distances, we have the *locality signature*, which is shown in Figure 1(b) for the example trace. For a fully-associative LRU cache, an access misses in the cache if and only if its reuse distance is greater than the cache size. Figure 1(c) shows all nonzero miss rates of the example execution on all cache sizes. In general, a locality signature captures the average locality of an

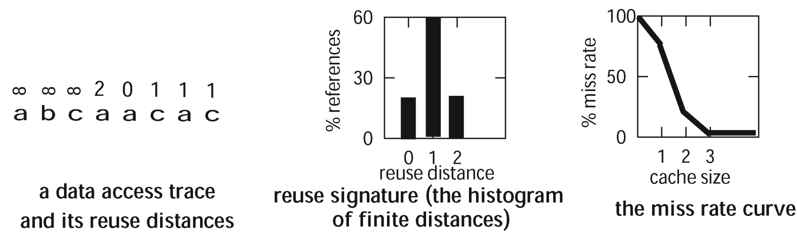


Fig. 1. Example reuse distances, locality signature, and miss rate curve.

execution from the view of the hardware as the miss rate in caches of all sizes and all levels of associativity [Mattson et al. 1970; Smith 1976; Hill and Smith 1989] and from the view of the operating system as the size of the working sets [Denning 1980].

At the program level, locality analysis is hampered by complex control flows and data indirection. For example, pointer usage obscures the location of the datum being accessed. With reuse distance, we can avoid the difficulty of code analysis by directly examining the execution or, more accurately, the locality aspect of the execution. Compilers may make local changes to a program, for example, by unrolling a loop. Modern processors, likewise, may reorder instructions within a limited execution window. These transformations affect parallelism but not cache locality. The unchanging locality cannot be seen in the reuse distance since the number and the length of long reuse distances stay the same with and without the transformations. As a direct measure, reuse distance is unaffected by coding and execution variations that do not affect locality.

Furthermore, reuse distance makes it possible to correlate data usage across training executions. Since a program may allocate different data (or the same data in different locations) between runs, we cannot directly compare data addresses, but we may find correlations in their reuse distances. More importantly, we can partition memory accesses by decomposing the locality signature into subcomponents with only short- or long-distance reuses. As we shall see, programs often exhibit consistent patterns across inputs, at least in some components. As a result, we can characterize whole-program locality by defining common patterns and identifying program components that have these patterns.

A major difficulty of training-based analysis is the immense size of execution traces. A small program may produce a long execution, in which a modern processor may execute billions of operations a second. Section 2 addresses the problem of measuring reuse distance. We present two approximate algorithms: one guarantees a relative precision and the other an absolute precision. Since data may span the entire execution between uses, a solution must maintain some representation of the trace history. The approximate solutions use a data structure called a *scale tree*, in which each node represents a time range of the trace. By properly adjusting these time ranges, an analyzer can examine the trace and compute approximate reuse distance in effectively constant time regardless of the length of the trace. Over the past four decades, there has been a

steady stream of solutions developed for the measurement problem. We review the other solutions in Section 2.3 and present a a new lower-bound result in Section 2.4.

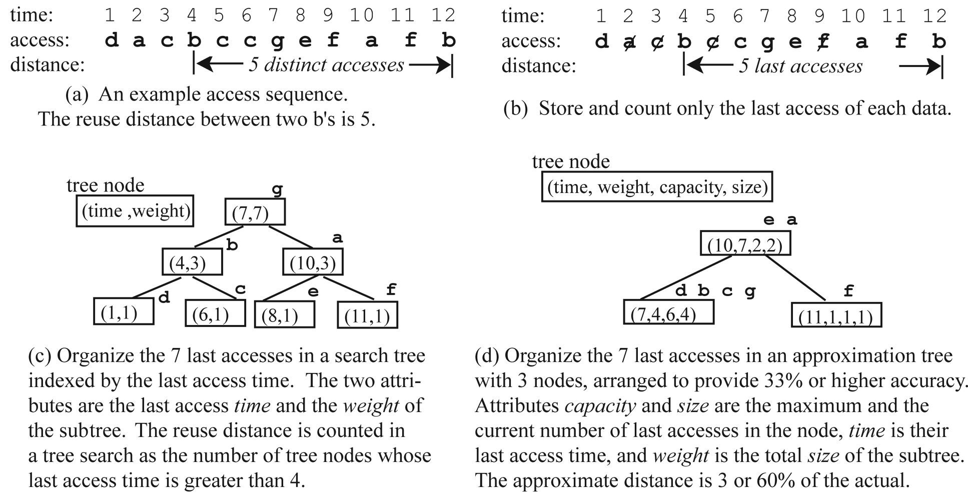
The key to modeling whole-program locality is prediction across program inputs. Section 3 describes the prediction process, which first divides data accesses into groups, then identifies statistical patterns in each group, and finally computes parameterized models that yield the least error. Pattern analysis is assisted by the fact that reuse distance is always bounded and can change at most as a linear function of the size of the data. We present five prediction methods assembled from different division schemes, pattern types, and statistical equations. Two methods are *single-model*, which means that a locality component, that is, a partition of memory accesses, has only one pattern. The other three are *multimodel*, which means that multiple patterns may appear in the same component. These offline models can be used in online prediction using a technique called distance-based sampling.

The new techniques of approximate measurement and statistical prediction are evaluated in Section 4 using real and artificial benchmarks. Section 4.1 compares eight analyzers and shows that approximate analysis is substantially faster than previous techniques in measuring long reuse distances. Section 4.2 compares five prediction techniques and shows that most programs have predictable components, and the accuracy and efficiency of prediction increase with additional training inputs and with multimodel prediction. On average, the locality in fifteen test programs can be predicted with 94% accuracy. Programs that are difficult to predict include interpreters and scientific code with high-dimension data. Interestingly, because reuse distance is execution-based, our analyses can reveal similarities in inherent data usage among applications that do not share code.

Our locality prediction techniques are examples of a broader approach we call *behavior-based program analysis*. Conventional program analysis identifies invariant properties by examining program code. Behavior analysis infers common patterns by examining program executions. Section 5 discusses related work in locality analysis using program code and behavior metrics including reuse distance, access frequency and data streams. Locality analysis has numerous uses in performance modeling, program improvement, cache and virtual memory management, and network caching. Section 6 presents a taxonomy that classifies the uses of reuse distance into five dimensions—program code, data, input, time, and environment. Many of these uses may benefit from the fast analysis and predictive modeling described in this article.

2. APPROXIMATE REUSE-DISTANCE MEASUREMENT

In our problem setup, a trace is a sequence of T accesses to N distinct data items. A reuse-distance analyzer traverses the trace and measures the reuse distance for each access. At each access, the analyzer finds the previous time the data was accessed and counts the number of different data elements accessed in between. To find the previous access, the analyzer assigns each access a logical time and stores the last access time of each datum in a hash table. In the worst



(a) An example access sequence. The reuse distance between two b's is 5.

(b) Store and count only the last access of each data.

(c) Organize the 7 last accesses in a search tree indexed by the last access time. The two attributes are the last access *time* and the *weight* of the subtree. The reuse distance is counted in a tree search as the number of tree nodes whose last access time is greater than 4.

(d) Organize the 7 last accesses in an approximation tree with 3 nodes, arranged to provide 33% or higher accuracy. Attributes *capacity* and *size* are the maximum and the current number of last accesses in the node, *time* is their last access time, and *weight* is the total *size* of the subtree. The approximate distance is 3 or 60% of the actual.

Fig. 2. An example illustrating the reuse-distance measurement. Part (a) shows a reuse distance. Parts (b) and (c) show its measurement by the Bennett-Kruskal algorithm and the Olken algorithm. Part (d) shows our approximate measurement with a guaranteed precision of 33%.

case, the previous access may occur at the beginning of the trace, the difference in access time is up to $T - 1$, and the reuse distance is up to $N - 1$. In large applications, T can be over 100 billion, and N is often in the tens of millions.

We use the example in Figure 2 to introduce two previous solutions and then describe the basic idea for our solution. Part (a) shows an example trace. Suppose we want to find the reuse distance between the two accesses of b at time 4 and 12. A solution has to store enough information about the trace history before time 12. Bennett and Kruskal [1975] discovered that it is sufficient to store only the last access of each datum, as shown in Part (b) for the example trace. The reuse distance is measured by counting the number of last accesses, stored in a bit vector rather than using the original trace.

The efficiency was improved by Olken [1981], who organized the last accesses as nodes in a search tree keyed by their access time. The Olken-style tree for the example trace has 7 nodes, one for the last access of each datum, as shown in Figure 2(c). The reuse distance is measured by counting the number of nodes whose key values are between 4 and 12. The counting can be done in a single tree search, first finding the node with key value 4 and then backing up to the root accumulating the subtree weights [Olken 1981]. Since the algorithm needs one tree node for each data location, the search tree can grow to a significant size when analyzing programs with a large amount of data.

While it is costly to measure long reuse distances, we rarely need the exact length. Often the first few digits suffice. For example, if a reuse distance is about one million, it rarely matters whether the exact value is one million or one million and one. Next we describe two approximate algorithms that extend the Olken algorithm by adapting and trimming the search tree.

The new algorithms guarantee two types of precision for the approximate distance, $d_{approximate}$, compared to the actual distance, d_{actual}. In both types, the

approximate distance is no greater than the actual distance. *Relative precision* means that the maximal error is no more than a constant fraction e of the actual distance. *Absolute precision* means that the maximal error b is a constant. Here the term "precision" means the portion of a value that can be reliably measured. We also use the term "accuracy" interchangeably. The formal definition of the two guarantees is as follows:

(1) Relative precision w/ max error e: $0 < e < 1$ and $0 \leq \frac{d_{actual} - d_{approximate}}{d_{actual}} \leq e$.
(2) Absolute precision w/ max error b: $b > 0$ and $0 \leq d_{actual} - d_{approximate} \leq b$.

Instead of using a tree node to store the last access of one data element as in the Olken algorithm, the approximate analysis uses a tree node to store a time range that may include the last accesses of multiple data elements. We call the new tree a *scale tree* and define the size of each node as the number of data elements last accessed in its time range. An example scale tree is shown in Figure 2(d), which stores the last accesses of 7 variables approximately in 3 tree nodes with sizes 2, 4, and 1, respectively (in comparison, the precise representation in Figure 2(c) requires 7 tree nodes). The size of the scale tree, measured by the number of tree nodes, equals N divided by the average node size. The error in approximation can be as large as the maximal node size. The Olken algorithm uses unit-size nodes and has full precision. The problem for the approximation algorithms is how to inflate the node size so the tree size is minimized while the measurement error is bounded.

In the following discussion, we do not consider the cost of finding the last access time. This can be performed by looking it up in a hash table, which has an $O(1)$ expected cost per access. The space cost is $O(N)$, although it can be reduced to a constant using multipass analysis [Bennett and Kruskal 1975].

2.1 Approximation with a Relative Precision

We describe the scale tree and its two types of operations. The first happens at every access to compute the reuse distance. The second happens periodically to compress the tree by coalescing the time ranges and reducing the number of tree nodes.

A node in a scale tree has 7 attributes, as defined in Figure 3. The time attribute is the end of its time range and also the search key. The size attribute is the number of data last accessed in the time range. The weight attribute is the total size of all the tree node's children. We assume the tree is a binary tree, so each node has left and right children. For the purpose of compression, we link tree nodes in a linear order using the *prev* attribute, by which each node is tied to the node of the immediately earlier time range. For example, the time range of node x is from $x.prev.time + 1$ to $x.time$. The last and most important attribute is *capacity*, which sets the upper bound of the node size and in turn determines the size of the tree and the precision of the approximation.

Let the current access be to datum d. The main routine, *ReuseDistance* shown in Figure 3, is called given as input the *last* and *current* access time. As the first step, it calls the subroutine *TreeSearchDelete*, which finds the host node

```
data declarations
    TreeNode = structure(time, weight, capacity, size, left, right, prev)
    root: the root of the tree
    e: the bound of the relative error

algorithm ReuseDistance(last, current)
    // inputs are the last and current access times
    TreeSearchDelete(last, distance)
    latest = new TreeNode(current, 1, 1, 1, ⊥, ⊥, ⊥)
    TreeInsert(latest)
    if (tree_size ≥ 4 ∗ log_{1/(1−e)} root.weight + 4)
        TreeCompress(latest) // removes at least half of the nodes
     end if
    return distance
end algorithm

subroutine TraceSearchDelete(last, distance)
    node = root
    distance = 0
    while true
      node.weight = node.weight − 1
      if (last < node.time and node.prev exists and last ≤ node.prev.time)
          if (node.right exists)
            distance = distance + node.right.weight
          if (node.left not exists) break
          distance = distance + node.size
          node = node.left
      elseif (last > node.time)
          if (node.right not exists) break
          node = node.right
      else  exit loop
      end if
    end while
    node.size = node.size − 1
end subroutine TreeSearchDelete
```

Fig. 3. Approximate analysis with a relative precision $1 - e$. Part I.

containing the last access of d and traverses the search path backward to calculate the approximate reuse distance using the sum of the subtree weights as in the Olken algorithm.

Since the last access of d is changed, *TreeSearchDelete* removes the last record by decrementing the *size* attribute of the host node and the *weight* attribute of all parent nodes. Subroutine *TreeInsert* is then called to add a new node into the tree representing the new last access, the current access. It rebalances the tree as needed. The insertion procedure is not shown since it depends on the type of search tree being used.

The loss of precision occurs at the host node. It contains a group of last accesses but we cannot know which is the last access of d. To prevent overestimating, we assume it is the last one in the group. The error is at most $size - 1$, which is at most $capacity - 1$, since $size \leq capacity$.

The initial capacity of a new node is 1. As tree nodes become dated, their capacity is adjusted by the subroutine *TreeCompression*, shown in Figure 4.

```
subroutine TreeCompress(n)
    // Initially n is the latest node in the tree
    distance = 0
    n.capacity = 1
    while (n.prev exists)
      if (n.prev.size + n.size ≤ n.capacity)
          // merge n.prev into n
          n.size = n.size + n.prev.size
          n.prev = n.prev.prev
          deallocate n.prev
      else
          distance = distance + n.size
          n = n.prev
          n.capacity = ⌊distance * e/(1−e)⌋ + 1
      end if
    end while
    Build a balanced tree from the list and update the root
end subroutine TreeCompress
```

Fig. 4. Approximate analysis with a relative precision $1 - e$. Part II.

It uses the *prev* link to traverse tree nodes in reverse chronological order and assigns the capacity of each node x to $distance * \frac{e}{1-e} + 1$, where e is the error bound ($0 < e < 1$), and *distance* is the number of distinct data accessed after x's time range. Since the maximal error at node x is $x.capacity - 1$, the maximal relative error is $\frac{x.capacity-1}{distance+x.capacity-1} = e$. *TreeCompression* will also merge adjacent time ranges as long as the combined size does not exceeds the capacity. The size of the tree is minimized for the error bound.

In the main routine, *ReuseDistance*, tree compression is triggered when the number of tree nodes exceeds the threshold $4 \log_{\frac{1}{1-e}} N + 4$, where N is the number of distinct elements that have been accessed. The following theorem shows that the compression always removes at least half of the tree nodes.

THEOREM 1. *For a trace of T accesses to N data elements, the approximate reuse distance measurement with a bounded relative error e $(0 < e < 1)$ takes $O(T \log^2 N)$ time and $O(\log N)$ space, assuming it uses a balanced tree.*

PROOF. Since the tree is compressed whenever it grows to $4 * \log_{\frac{1}{1-e}} N + 4$ nodes, the number of tree nodes cannot exceed $O(\log N)$. We next show that every time it is invoked, *TreeCompress* removes at least half of the tree nodes.

Since the compression routine marches backward in time, we number the compressed nodes in reverse chronological order as $n_0, n_1, \ldots$, and n_r, with n_0 being the latest node. Assume r is an odd number (if r were even, we could add a zero-size node). Consider each pair n_{2i} and n_{2i+1}, $i = 0, \ldots, \frac{r-1}{2}$. Let $size_i$ be the combined size of n_{2i} and n_{2i+1} and $sum_i = \sum_{j=0,\ldots,i} size_j$ be the total size of nodes up to and including n_{2i+1}.

Since the capacity of the node n_{2i} is set to $\lfloor sum_{i-1} * \frac{e}{1-e} \rfloor + 1$ by the algorithm, the combined size of the node pair, $size_i$, must be at least $\lfloor sum_{i-1} * \frac{e}{1-e} \rfloor + 2$; otherwise the ith node pair should have been merged into a single node. We now have $size_0 \geq 1$ and $size_i > sum_{i-1} * \frac{e}{1-e}$. Since $sum_i = size_i + sum_{i-1}$, by

induction we have $sum_i > (1+\frac{e}{1-e})^i$ or $i < \log_{\frac{1}{1-e}} sum_i$. Let $M_{compressed}$ be the size of the tree after compression. Since $M_{compressed} = r + 1 \leq 2i + 2$ and $sum_i = N$, $M_{compressed} < 2 * \log_{\frac{1}{1-e}} N + 2$. Comparing to the starting size, we see that each compression must cut out half of the tree nodes.

Now we consider the time cost. Assume that the tree is balanced and has M tree nodes ($M \leq 4 \log_{\frac{1}{1-e}} N + 4$). The time for the tree search, deletion, and insertion is $O(\log M)$ per access. Tree compression happens periodically after a tree growth of at least $2 \log_{\frac{1}{1-e}} N + 2$ or $M/2$ tree nodes. Since one tree node is added for each access, the number of accesses between successive tree compressions is at least $M/2$ accesses. Each compression takes $O(M)$ time because it examines each node in a constant time, and the tree construction from an ordered list takes $O(M)$ time. Hence the amortized compression cost is $O(1)$ for each access. The total time is therefore $O(\log M + 1)$, or $O(\log^2 N)$ per access. □

2.2 Approximation with Absolute Precision

For a cut-off distance c and a constant error bound b, the absolute-precision algorithm divides the access trace into two parts: the *precise trace* records the last c elements accessed, and the *approximate trace* stores older accesses in a tree where the capacity of each tree node is set to $b + 1$. As a result, the measurement is accurate for reuse distances up to c and approximate for larger distances with an error no more than b. Periodically, the algorithm transfers data from the precise trace to the approximate trace.

We have described a detailed algorithm and its implementation using a B-Tree for both the precise and approximate trace [Zhong et al. 2002]. Here we generalize it to a class of algorithms. The precise trace can use a list, a vector, or any type of tree, and the approximate trace can use any type of tree, with two requirements. First, the size of the precise trace is bounded by a constant. Second, a minimal occupancy of each tree node is guaranteed. To satisfy the first requirement, we transfer the last accesses of c data elements from the precise trace to the approximate trace when the size of the precise trace exceeds $2c$. To ensure minimal occupancy, we merge two consecutive tree nodes if their total size falls below the capacity of the succeeding node. The merge operation guarantees at least half utilization of the capacity b at each node. Therefore, the number of nodes in the approximate tree is at most $\frac{2N}{b}$.

We have implemented a splay tree [Sleator and Tarjan 1985] version of the algorithm and will use only the approximate trace ($c = 0$) in the analyzer for runtime locality analysis (i.e., distance-based sampling in Section 3.5) because the analyzer has the fastest speed, as shown later in Section 4.1.

2.3 Comparison with Related Concepts and Algorithms

Mattson et al. [1970] showed that buffer memory could be modeled as a stack, if the method of buffer management satisfied the *inclusion* property in that a smaller buffer would hold a subset of data held by a larger buffer. They showed that the inclusion property is satisfied when a buffer is managed by common replacement policies including least recently used (LRU), least frequently used

Table I. The Asymptotic Complexity of Reuse-Distance Measurement

Measurement Algorithms	Time	Space
trace as a stack (or list) [Mattson et al. 1970]	$O(TN)$	$O(N)$
trace as a vector (interval tree) [Bennett and Kruskal 1975; Almasi et al. 2002]	$O(T \log T)$	$O(T)$
trace as a search tree [Olken 1981] [Sugumar and Abraham 1993; Almasi et al. 2002]	$O(T \log N)$	$O(N)$
aggregate counting [Kim et al. 1991]	$O(Ts)$	$O(N)$
approximation using time [Berg and Hagersten 2004; Shen et al. 2007]	$O(T)$	$O(1)$
approx. w/ relative precision	$O(T \log^2 N)$	$O(\log N)$
approx. w/ absolute precision	$O(T \log \frac{N}{b})$	$O(\frac{N}{b})$

T is the length of execution, N is the size of program data, s is the number of (measured) cache sizes, b is the error bound.

(LFU), optimum (OPT), and a variant of random replacement. They defined a collection of *stack distances*. These concepts formed the basis of storage system evaluation and enabled much of the experimental research in virtual memory management in the subsequent decades.

Stack distance is also used extensively in studies of cache memory. But it is not a favorable metric in low-level cache design because it does not model issues such as write-backs, cache-line prefetch, and queuing delays. Some of the drawbacks have been remedied by techniques that modeling the effect of set associativity [Smith 1976; Hill and Smith 1989] and write-backs and subblocks for fully associative [Thompson and Smith 1989] and set-associative caches [Wang and Baer 1991].

Reuse distance is the same as the LRU stack distance. It is informative to use the shorter name here because our primary purpose is program analysis. Locality as a program property exists without the presence of buffer memory or caches, so the notion of the stack is immaterial. In addition, reuse distance can be measured directly and much more quickly using a tree (or a bit vector) instead of a stack.

Since 1970, there have been steady improvements in reuse distance measurement. We categorize previous methods by their organization of the trace. The first three rows of Table I show methods using a stack, a bit vector, and a tree. Mattson et al. [1970] gave the first algorithm, which used a stack. Bennett and Kruskal [1975] observed that a stack was too slow to measure long reuse distances in database traces. They used a bit vector and built an m-ary interval tree on it. They also showed how to make the hash table smaller using multi-pass analysis. Olken [1981] gave the first tree-based algorithm. He also showed how to compress the bit vector and improve the Bennett-Kruskal algorithm to the efficiency level of his tree-based algorithm. Sugumar and Abraham [1993] showed that a splay tree [Sleator and Tarjan 1985] had better memory performance and developed a widely used cache simulator, *Cheetah*. Almasi et al. [2002] showed that by recording the empty regions instead of the last accesses in the trace, they could improve the efficiency of vector and tree based methods by 20% to 40%. They found that the modified Bennett-Kruskal algorithm was faster than the Olken algorithm with AVL or red-black trees.

Kim et al. [1991] gave an algorithm that stores the last accesses in a list and embeds markers for cache sizes. It measures the miss rate precisely but not the reuse distance. The space cost is proportional to the largest cache size, which is N if we measure for caches of all sizes. Instead of reuse distance, the access distance, that is, the logical time between the two consecutive accesses to the same datum, has been used to estimate the miss rate for caches of all sizes in StatCache [Berg and Hagersten 2004, 2005] and to statistically infer the reuse distance in time-based prediction [Shen et al. 2007]. The two statistical techniques have a linear time cost but do not guarantee the precision of the result. In addition, Zhong and Chang [2008] used sampling analysis to reduce the constant factor in the cost of reuse-distance measurement.

The literature on algorithm design has two related problems: finding the number of distinct elements in a sequence of m elements each of which is between 0 and n, and finding the number of 1's in a window of m binary digits. The goal of streaming algorithms is to solve these problems incrementally without storing the entire sequence. Alon et al. [1996] gave a simple proof (Proposition 3.7) of the previously known result that any such algorithm must use $\Omega(n)$ memory bits. For counting the number of 1's over a sliding window of size m, Datar et al. [2002] gave a deterministic algorithm with optimal space complexity $O(\log^2 m)$ bits. They extended it to count the number of distinct values in a sliding window "with an expected relative accuracy of $O(\frac{1}{\sqrt{n}})$ using $O(n \log^2 m)$ bits of memory", based on probabilistic counting [Flajolet and Martin 1983]. The relative precision algorithm in this paper can solve the same sliding-window problem deterministically with constant relative precision using $O(n \log m)$ bits in the hash table and $O(\log n \log m)$ bits in the scale tree.

The approximate measurement is asymptotically faster than exact algorithms. The space cost of the search tree is reduced from linear to logarithmic. The time cost per access, $O(\log^2 N)$, is effectively constant for any practical data size N. The improvement is important when analyzing a program at the data-element granularity like we do in program locality analysis. Next we show a lower bound result for the space cost, which suggests that approximation is necessary to obtain this level of efficiency.

2.4 A Lower Bound Result

The following theorem gives the minimal space needed by an exact algorithm.

THEOREM 2. *The space cost for accurately measuring reuse distance is* $\Omega(N \log N)$ *bits, where N is the largest reuse distance.*

PROOF. The trace may contain accesses to N distinct data elements. Assuming prior to a logical time k, all N elements have been accessed, the reuse distance at k depends on the relative order of the last accesses of N elements. The number of possible orders is the number of permutations of N elements or $N!$.

An accurate method must be able to distinguish between any two different permutations. Otherwise, let us assume that there exists an accurate measurement that does not distinguish between two permutations Q and R, and datum x is last accessed at a different point in Q than in R. If given the two traces Qx

and Rx, the method would not be able to show that the reuse distance of the last access (of x) is different in the two traces. This contradicts the assumption that the method is accurate. Since an exact method must distinguish between all $N!$ permutations, it must store $\Omega(\log N!)$ or $\Omega(N \log N)$ bits. □

The lower bound result has two significant implications. First, the $\Omega(N \log N)$ lower bound differs from the $\Omega(N)$ lower bound of counting the number of 1's in a sliding window [Alon et al. 1996], so the problem of reuse distance measurement is inherently harder than the sliding window problem in streaming. Second, we observe that in the method of Olken [1981], both the hash table and the search tree have $O(N)$ entries of $O(\log T)$ bits per entry, so the space cost, $O(N \log T)$ bits, is close to optimal. To match the time efficiency of the approximate algorithm, an exact algorithm must process $O(N)$ items of information in $O(\log^2 N)$ steps for each access, which seems improbable. Hence the lower bound result suggests that we may not improve exact measurement much beyond Olken's result. For a greater efficiency we may have to resort to approximation, as we have done using the scale tree.

3. LOCALITY PREDICTION

Locality prediction has three steps: dividing reuse distances into groups, analyzing their length in training executions, and constructing a statistical model to predict their length in all executions. The only parameter of the model is the *input size*. In Section 3.5, we define the input size computationally using a technique called *distance-based sampling*. In most cases it is equivalent to N, the size of the data touched by an execution. We therefore use the terms input size and program data size interchangeably.

3.1 Decomposing the Locality Signature

As we divide reuse distances into groups, it is desirable to control the range of reuse distances in a group and the size of the group. Metaphorically speaking, the range and the size can be considered the two dimensions that control an inspection lens's resolution. The range should not be too large because it may include reuse distances representing different locality. The size should not be too small because it would increase the computational cost without improving accuracy.

We represent the locality signature using two types of histograms. In a *reuse-distance histogram* (or *distance histogram*), the x-axis gives the length of reuse distance in consecutive ranges or bins, and the y-axis shows the percentage of all reuse distances that fall in each range. For each bin in the histogram, we call the range of reuse distances its *width* and the frequency of reuse distances its *size*. The width may grow in a *linear scale*, for example, $[0, 2k), [2k, 4k), [4k, 6k), \cdots$; a *logarithmic scale*, for example, $[0, 1), [1, 2), [2, 4), [4, 8), \cdots$; or a *log-linear scale*, for example, the ranges below 2048 are logarithmic and the rest are linear. Figure 5(a) shows the logarithmic scale histogram of a fluid dynamics simulation program.

Alternatively, we can sort all reuse distances, divide them into equal-size partitions, line the groups up along the x-axis, and show the average reuse

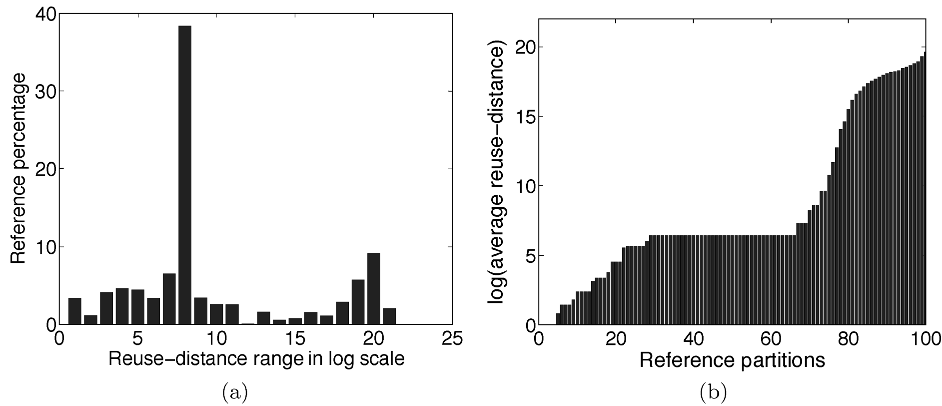


Fig. 5. Example histograms for program *SP* with input size 28^3. (a) The log-scale distance histogram shows the percentage of reuse distances (the y-axis) that fall into ranges of base-2 logarithmic scale (the x-axis). (b) The reference histogram shows the average reuse distance (the y-axis) for each 1% of reuse distances sorted by increasing length (the x-axis).

distance of each group on the y-axis. We call this a *reference histogram* and each bin a *reference partition*. Figure 5(b) shows the reference histogram in 100 partitions for the same example program. The first bin shows that the average length is 0 for the shortest 1% of reuse distances. The two histograms can be explained using nomenclature from probability theory. If we view reuse distance as a random variable, the distance histogram, for example, Figure 5(a), is the density function, and the reference histogram, for example, Figure 5(b), is the transpose of the cumulative density function.

The two types of histograms have complementary properties for behavior decomposition. With distance histograms, we can easily control the range of the reuse distances in each group but not the size of the group. With reference histograms, all groups have the same size but the range of reuse distances in a group can be arbitrarily large.

For locality prediction, the reference histogram has two important advantages over the distance histogram. First, it isolates the effect of nonrecurrent computations such as the initialization code before the main computation loop. When the input size is sufficiently large, the effect of the nonrecurrent computation diminishes into a single partition in the reference histogram. The second is to balance between information loss and modeling efficiency. When many reuse distances have a similar length, the reference histogram may divide them to increase precision. When a few reuse distances cover a wide spread in length, the reference histogram uses large ranges to reduce the number of groups. The size of the group determines the granularity and the cost of prediction. A group size of 1% means that we analyze only 100 bins, and the error in a bin does not affect more than 1% of the overall accuracy.

In our implementation, we generated the distance histogram using a log-linear scale. The bins' sizes were powers-of-2 up to 2048 and each remaining bin had a size of 2048. We compute the average distance of each bin and use it to convert the log-linear distance histogram to the reference histogram. Our

reference histogram has 1000 bins. We will evaluate prediction accuracy using the three types of histograms: the reference histogram, the log-linear distance histogram, and the logarithmic distance histogram. The last type is usually one or two orders of magnitude more compact than the first two types.

3.2 Constant, Linear, and Sublinear Patterns

Patterns are defined for each group of reuse distances. Let the groups be $\langle g_1, g_2, \ldots, g_B \rangle$ for one execution and $\langle \hat{g}_1, \hat{g}_2, \ldots, \hat{g}_B \rangle$ for another, where B is the number of groups. Let the average reuse distances of g_i and $\hat{g}_i$ be d_i and $\hat{d}_i$. Let s and $\hat{s}$ be the input size of the two executions. We find the closest linear function that maps the input size to the reuse distance. Specifically, we find the two coefficients, c_i and e_i, that satisfy the following two equations.

$$d_i = c_i + e_i * f_i(s) \tag{1}$$

$$\hat{d}_i = c_i + e_i * f_i(\hat{s}), \tag{2}$$

where f_i is the pattern function. Once we define common patterns f_i, the problem becomes one of linear regression.

Since *the largest reuse distance cannot exceed the size of program data*, the pattern function f_i can be at most linear and cannot be a general polynomial function. We consider the following five choices of f_i:

$$0; \;\; s; \;\; s^{1/2}; \;\; s^{1/3}; \;\; s^{2/3}.$$

We call the first, 0, the *constant pattern*. A group of reuse distances has a constant pattern if their average length does not change with the input. We call the second, s, the *linear pattern*. A bin i has a linear pattern if the average distance changes linearly with the program input size, i.e. $\frac{d_i - c_i}{\hat{d}_i - c_i} = e_i \frac{s}{\hat{s}}$, where c_i and e_i are constants. Constant and linear patterns are the lower and upper bound of the reuse distance changes. Between them are three sub-linear patterns. The pattern $s^{1/2}$ happens in two-dimensional problems such as matrix computations. The other two happen in three-dimensional problems such as ocean simulation. We could consider higher dimensional problems in the same way, although we did not find a need in our test programs.

3.3 Single-Model Prediction

In single-model prediction, each group has a single pattern. For a group of reuse distances, we calculate the ratio of their average distance in two executions, $d_i/\hat{d}_i$, and pick f_i to be the pattern function that is closest to $d_i/\hat{d}_i$. We take care not to mix sublinear patterns from a different number of dimensions. In our experiments, the dimensionality was given as an input to the analyzer. This can be automated by trying all choices and using the best fit.

Using more than two training inputs may produce a better prediction, because more data may reduce the noise from imprecise reuse distance measurement and histogram construction. We consider more inputs as follows. For each bin, instead of two linear equations, we have as many equations as the number of training runs. We use least square regression to determine the best values

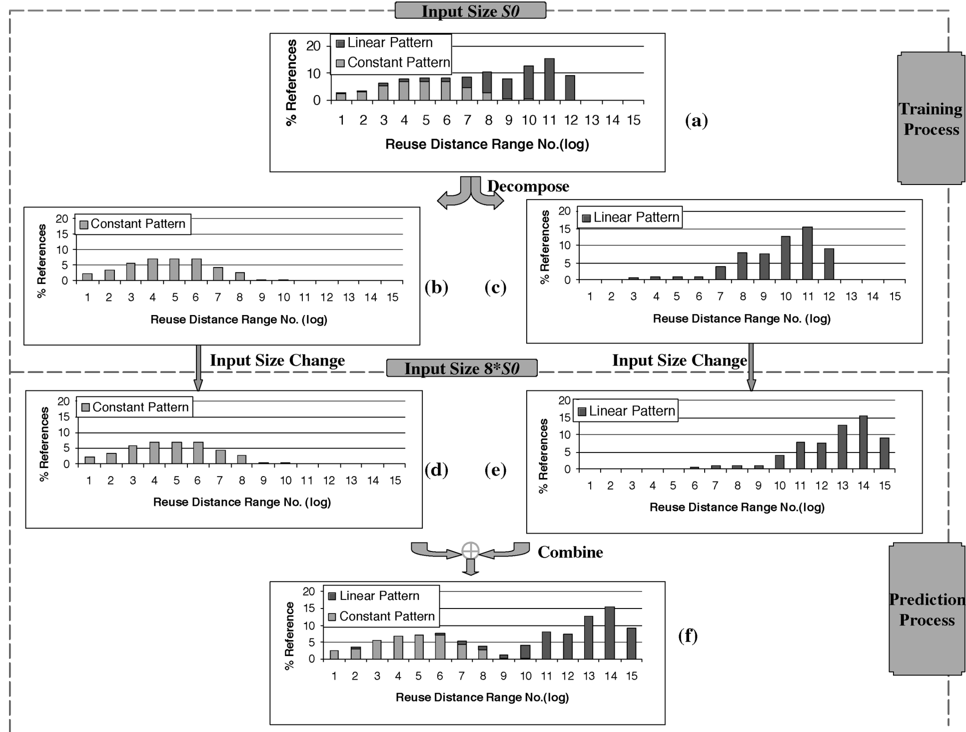


Fig. 6. An example of multimodel prediction. Part (a) is the standard histogram of input s_0. Part (b) and (c) show the composition of the constant and linear patterns in the standard histogram. Given a new input $8 * s_0$, the constant part remains unchanged, shown in (d). The distance of the linear part increases by a factor of eight, shown in (e). The prediction combines (d) and (e) to produce (f).

for the two unknowns. We will evaluate the relation between the number of training inputs and the prediction accuracy.

3.4 Multimodel Prediction

Modelmodel prediction allows a group of reuse distances to have mixed patterns. For example, some fraction of a group has one pattern, and the rest has a different pattern. In multimodel prediction, the size of the ith group, $h_i(s)$, is as follows.

$$h_i(s) = \varphi_{m_1}(s, i) + \varphi_{m_2}(s, i) + \cdots + \varphi_{m_j}(s, i), \qquad (3)$$

where s is the size of the input, and $\varphi_{m_1} \cdots \varphi_{m_j}$ are all possible pattern functions.

To ground the calculation on a single basis, we arbitrarily pick the result of one of the training runs as the *standard histogram*. In single-model prediction, one group in one histogram corresponds to one group in another histogram. In multimodel prediction, one group in one histogram may correspond to a piece in every group in another histogram.

We illustrate the process of multimodel prediction through an example in Figure 6. The standard histogram is shown in Part (a). The size of its input

is s_0. Other histograms are not shown, although they are used by the analysis to compute the mixing of patterns in the standard histogram. The standard histogram has 12 bins, and each bin has two models—the constant and the linear pattern. The two patterns are separated into two pieces shown in Part (b) and (c). Given another input size, $8 * s_0$, we predict the reuse distance according to the patterns. The constant pattern remains unchanged, shown in Part (d). The distance in the linear pattern is octupled by moving the bars right by 3 units along the x-axis, shown in Part (e). Finally, the prediction process combines the constant and linear pieces and produces the predicted histogram for input size $8 * s_0$ in Part (f).

We show how to derive the composition of patterns in the standard histogram, again through an example. Let s_0 be the size of the standard input, and $s_1 = 3s_0$ be the size of another training input. Let $\varphi(s, r)$ be the portion of reuse distances in range r at input s. For this example we again assume only two patterns and use φ_c for the constant pattern and φ_ℓ for the linear pattern. We use logarithmic scale ranges. The first four are $[0, 1\rangle, [1, 2\rangle, [2, 4\rangle, [4, 8\rangle$. The analysis assumes that reuse distances are distributed over a range continuously.

We compute the composition of the range $[4, 8\rangle$ in the histogram of $s_1 = 3s_0$ from the standard s_0 histogram as follows. The size of the bin $[4, 8\rangle$ in histogram s_1 consists of constant and linear parts. The size of the constant part is the same in s_1 as in s_0. The size of the linear part comes from the range $[\frac{4}{3}, \frac{8}{3}\rangle$ in s_0. The relations are shown in the next three equations.

$$\begin{aligned}
\varphi(s_1, [4, 8\rangle) &= \varphi_c(s_1, [4, 8\rangle) + \varphi_\ell(s_1, [4, 8\rangle) \\
\varphi_c(s_1, [4, 8\rangle) &= \varphi_c(s_0, [4, 8\rangle) \\
\varphi_\ell(s_1, [4, 8\rangle) &= \varphi_\ell\left(s_0, \left[\frac{4}{3}, \frac{8}{3}\right\rangle\right) = \varphi_\ell(s_0, \left[\frac{4}{3}, 2\right\rangle) + \varphi_\ell\left(s_0, \left[2, \frac{8}{3}\right\rangle\right).
\end{aligned}$$

We assume the reuse distance has uniform distribution in each bin. Hence,

$$\begin{aligned}
\varphi_\ell\left(s_0, \left[\frac{4}{3}, 2\right\rangle\right) &= \left(\frac{2 - 4/3}{2 - 1}\right)\varphi_\ell(s_0, [1, 2\rangle) = \frac{2}{3}\varphi_\ell(s_0, [1, 2\rangle) \\
\varphi_\ell\left(s_0, \left[2, \frac{8}{3}\right\rangle\right) &= \left(\frac{8/3 - 2}{4 - 2}\right)\varphi_\ell(s_0, [2, 4\rangle) = \frac{1}{3}\varphi_\ell(s_0, [2, 4\rangle).
\end{aligned}$$

Therefore,

$$\varphi(s_1, [4, 8\rangle) = \varphi_c(s_0, [4, 8\rangle) + \frac{2}{3}\varphi_\ell(s_0, [1, 2\rangle) + \frac{1}{3}\varphi_\ell(s_0, [2, 4\rangle).$$

After processing each bin of all training inputs in a similar manner, we obtain an equation group. The unknown variables are the size of the patterns in the standard histogram. Regression techniques are used to find the mixing that fits training results with the least error.

3.5 Distance-Based Sampling

Distance-based sampling is a heuristic for quickly estimating the input size by analyzing only the beginning of an execution. It takes samples of long reuse

distances and selects one to represent the input size. The rationale behind this scheme is the assumption that the change in input size is often proportional to the change in long reuse distances.

The sampling analysis uses a reuse-distance analyzer to monitor long-distance reuses. When a reuse distance is above a *qualification threshold*, the accessed memory location is taken as a data sample. Subsequent accesses to a data sample are recorded as access samples if the reuse distance is over a *temporal threshold*. To avoid picking too many data samples, it requires that a new data sample be at least a certain spatial distance away in memory from existing data samples. This is the *spatial threshold*. The sampling scheme requires certain manual effort to select the three thresholds for each program, although the threshold selection can be automated [Shen et al. 2007].

In a sequence of access samples, we define a *peak* as a time sample whose value is greater than that of its preceding and succeeding time samples. The analysis records the first k peaks of the first m data samples. A user evaluates these peaks in locality prediction and chooses the best one to represent the input size. The choice is program dependent but identical for all executions of the same program.

For most programs we have tested, it is sufficient to take the first peak of either the first or the second data sample. In one program, *Apsi*, all executions initialize the same amount of data but use a different amount in computation. We use the second peak as the input size. In some other programs, early peaks do not show a consistent relation with the input size, or the best peak appears near the end of an execution. We identify these cases during training and instruct the predictor to predict only the constant pattern.

Distance-based sampling can enable online prediction for an unknown input as follows. It first builds the offline model parameterized by the input size. When the execution of the test input starts, the sampling tool creates a twin copy of the program to collect the reuse distances. The sampled version runs in parallel with the original version until it detects the input size. For sampling to work, it requires that the input of the program be replicated, and that the sampled version not produce side effects.

3.6 Limitations

Although the analysis can handle any sequential program, this generality comes with several limitations. For programs with high-dimensional data, current pattern prediction requires that the shape of the data be similar in training and prediction. It should be possible to combine the pattern analyzer with a compiler and incorporate the shape of the data as parameters in the locality model. Note that locality prediction is useful only if the program is too complex for compiler analysis; otherwise, compiler analysis should be used or combined with locality prediction (see Section 5 for a review of related techniques). An important assumption in locality prediction is that the percentage size of a group of reuse distances is the same in all executions of a program. For example, the group of the 1% shortest reuse distances in one execution corresponds to the group of the 1% shortest reuse distances in other executions of the same

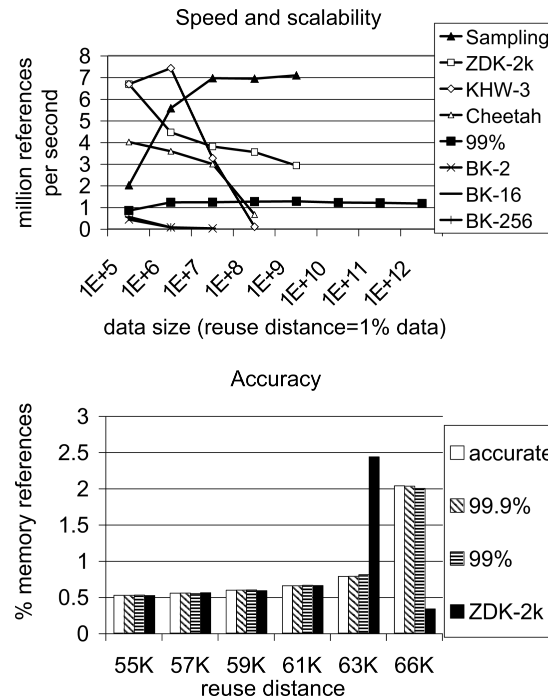


Fig. 7. A comparison of eight reuse-distance analyzers.

program. There is no logical reason that this relation has to hold in a program. We will use empirical validation by examining the accuracy of the prediction for a wide range of test programs. Finally, predicting locality does not mean predicting execution speed or execution time. The prediction gives the percentage of cache misses but not the effect on overall performance nor the total number of cache misses.

4. EVALUATION

For program analysis, we measure and predict reuse distance at the granularity of data elements. Analyzing data access at the finest granularity requires the highest efficiency and precision. The result shows the temporal locality independent of data layout. The same methods can be used to analyze temporal and spatial locality at larger data granularity such as cache blocks and memory pages (see Section 6 for a review of such studies).

4.1 Reuse Distance Measurement

Figure 7 compares the speed and accuracy of eight analyzers based on the algorithms described in Section 2. *Cheetah* [Sugumar and Abraham 1993] implements the Olken algorithm using a splay-tree. *BK-2*, *BK-16*, and *BK-256* are the bit-vector algorithm by Bennett and Kruskal [1975], implemented using k-ary trees with $k = 2, 16, 256$. These four measure reuse distance accurately. *KHW* is our implementation of Kim et al. [1991] with three markers at distances of 32, 16K, and the size of analyzed data. It classifies each reuse distance in three

bins. We test three approximate analyzers. *99%* is the relative-precision approximation with 99% accuracy. *Sampling* and *ZDK-2k* are absolute-precision approximations with maximal error $b = 2048$. *Sampling* uses a splay tree and only an approximate trace. *ZDK-2k* uses a B-tree and a mixed trace [Zhong et al. 2002]. The test program traverses N data elements twice with reuse distance equal to $N/100$. To measure only the cost of reuse-distance analysis, the hashing step is bypassed by pre-computing the last access time in all analyzers (except for *KHW*, which does not need the access time). The programs are compiled using *gcc* with full optimization (flag *-O3*) and tested on a 1.7 GHz Pentium 4 PC with 800 MB main memory.

Among the five accurate analyzers, the bit-vector methods are the slowest, *Cheetah* achieves an initial speed of 4 million memory references per second, and *KHW* with three markers is fastest (7.4 million memory references per second) for small data sizes. The accurate analyzers start to run out of the physical memory at 100 million data elements, so the three approximate analyzers become the fastest, with *Sampling* at 7 million references per second, *ZDK-2k* over 3 million references per second, and *99%* over 1 million references per second. *Sampling* and *ZDK-2k* do not analyze beyond 4 billion data elements since their implementation uses 32-bit integers.

Among the eight, the 99% precise approximate analyzer shows the most scalable performance. We use 64-bit integers in the program and test it for up to 1 trillion data elements. The asymptotic cost, $O(\log^2 N)$ per access, should be effectively linear in practice. We tested data sizes up to the 1 trillion because it is in the order of the length of a light year measured in miles. In the experiment, the analyzer ran at a near constant speed of 1.2 million references per second from 100 thousand data elements to 1 trillion data elements. The consistent high speed is remarkable considering that the data size and reuse distance differ by eight orders of magnitude. The speed was so stable that we could predict how much time our tests would take.

The lower graph of Figure 7 compares the accuracy of the approximation on a partial histogram of *FFT*. The y-axis shows the percentage of memory references, and the x-axis shows the distance on a linear scale between 55K and 66K with an increment of 2048. The *99.9%* and *99%* analyzers produce histograms that closely match the accurate histogram. The overall error is about 0.2% and 2% respectively. The analyzer with the constant error bound 2048, shown by the histogram marked ZDK-2k, misclassifies under 4% of the memory references at the far end of the histogram. If we compare the space overhead, accurate analyzers need 67 thousand tree or list nodes, *ZDK-2k* needs 2080 tree nodes (of which 32 nodes are in the approximate tree), *99.9%* needs 5869, and *99%* needs 823. The results show that approximate analyzers can greatly reduce the space cost without a significant loss of precision, and the cost and the accuracy are adjustable.

4.2 Locality Prediction

We begin by testing program locality prediction using reference histograms with 1000 bins, first for one benchmark program and then for all 15 programs.

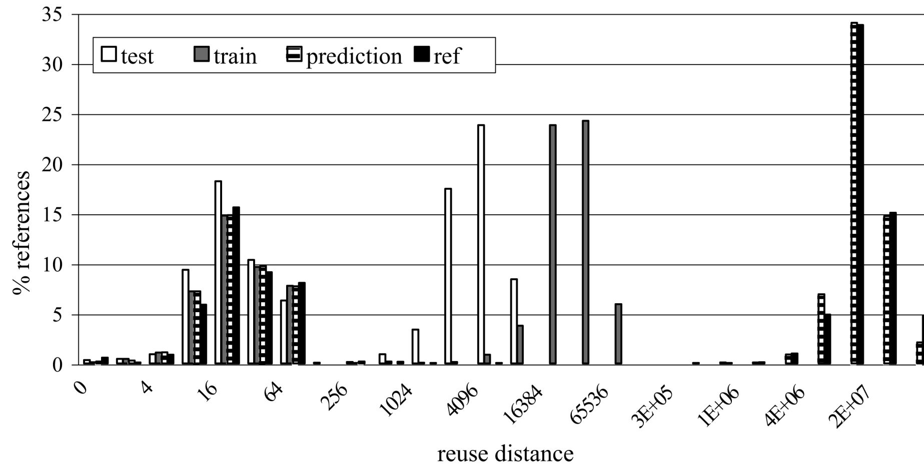


Fig. 8. Program locality prediction for *Spec2K/Lucas*.

Then we compare our full set of prediction methods and finally discuss a few notable features of whole-program locality analysis.

4.2.1 *Single-Model Prediction Based on Reference Histograms.* An illustrative example is the program *Lucas* from the SPEC 2000 benchmark suite. Based on the Lucas-Lehmer lemma, it tests the primality of very large numbers—numbers up to 2^{1000}. The program performs many large-number multiplications through specialized fast Fourier transforms coded using the C language. The program is difficult for a compiler to analyze.

The SPEC 2000 benchmark suite provides three inputs for the program. The smaller two are "test" and "train" inputs. Respectively they make 5 million reuses of 6 thousand data elements and 40 million reuses of 41 thousand data elements. The first two sets of bars in Figure 8 show their locality signatures in logarithmic scale distance histograms. The bars in the left half of their signatures show a similar distribution of short reuse distances. The bars in the right half show much longer reuse distances in "train" than in "test."

Single-model locality prediction measures the log-linear distance histogram for the two inputs, partitions the reuse distances into 1000 reference partitions, identifies constant and linear patterns, and builds a locality model parameterized by the input size measured using distance-based sampling. To test on-line prediction, we ran the third "ref" input. The execution has 644 billion accesses to 21 million data elements. After 0.4% of the execution time, distance-based sampling detected the input size. Substituting this in the model, we predicted the locality signature shown by the third group of bars in Figure 8. To compare, we measured the locality signature for the entire "ref" run, shown by the fourth set of bars.

Comparing the last two sets of bars in Figure 8, we see that the prediction largely agrees with the measurement. The two signatures match. The right half of the "ref" signature has no overlap with "test" and "train" signatures, yet the predicted signature is correct in shape and height, demonstrating the

ability by our method to predict large-scale behavior changes across the input of this program. The "ref" execution is four orders of magnitude longer and uses three orders of magnitude more data than "train" and "test" combined. The accurate prediction shows that the model is successful in characterizing the locality property at the program level, not just in a few executions.

We define prediction *accuracy* as follows. Let x_i and y_i be the size of ith bar in predicted and measured histograms. The accuracy is

$$accuracy = 1 - \frac{\sum_i |y_i - x_i|}{2}.$$

It measures the overlap between the two signatures, which ranges from 0% for no match to 100% for a complete match. For example, the prediction of *Lucas* in Figure 8 is 95% accurate.

Tables II and III list all the programs in our test set and summarize the accuracy and coverage of the single-model prediction. The test set consists of 15 benchmarks, including 9 floating-point programs and 6 integer programs. All programs came from SPEC 1995 and SPEC 2000 benchmark suites except for *SP* from the NAS benchmark suite and a textbook version of a two-dimensional *FFT* kernel. In experiments, we reduced the number of iterations in a program if it did not affect the overall pattern. Most experiments used DEC Alpha systems. We compiled the test programs with the DEC compiler using the default optimization (*-O3*). We used Atom [Srivastava and Eustace 1994] to instrument the binary code to collect the addresses of all loads and stores and fed them to our analyzer. The tool treated each distinct memory address as a data element.

The two tables have the same format, reporting each program in one row. The first two columns give the name and a short description of the program. The next column lists its reuse distance patterns, which can be constant, linear, or sublinear. Floating-point programs generally have more patterns than integer programs do. The fourth column shows the inputs used. They are all different as shown in the next three columns in terms of the number of distinct data elements, the number of data reuses per element, and the average reuse distance. The programs are listed in decreasing order of the average reuse distance.

Most inputs we used were standard test, train, and reference inputs from SPEC, with the following exceptions. For *GCC*, we picked the largest and two random inputs from the 50 files in its "ref" directory. *Tomcatv* and *Swim* had only two different data sizes. We added more inputs. The test input of *Twolf* had 26 cells and was too small. We randomly removed half of the cells in its train data set to produce a test input of 300 cells. *Applu* had a long execution time, so we replaced the reference input with a smaller one. Finally, the inputs of *Apsi* used high-dimensional data of different shapes, for which our predictor could not make an accurate prediction. We changed the shape of its largest input. We should also mention that all inputs of *Hydro2d* had a similar data size, but we did not make any change. *SP* and *FFT* did not come from SPEC, so we randomly picked their input sizes.

Columns 5 to 7 of the two tables show a range of data sizes from 14 thousand to 36 million data elements, average reuse frequency from 6 to over 300 thousand reuses per element, and average reuse distance from 15 to over

Table II. Prediction Accuracy and Coverage for Nine Floating-Point Programs

Benchmark	Description	Patterns	Inputs	Num. data elem.	Avg. reuses per elem.	Avg. dist. per elem.	Accuracy w/ data size (%)	Accuracy w/ sample size (%)	Cover-age (%)
Lucas	Lucas-Lehmer test	const	ref	20.8M	621	2.49E-1	**85.0**	**95.1**	**99.6**
(Spec2K)	for primality	linear	train	41.5K	971	2.66E-1	**85.9**	**81.8**	**100**
			test	6.47K	619	2.17E-1			
Applu	solution of five	const	45^3	9.33M	153	1.62E-1	**91.9**	**92.1**	**99.4**
(Spec2K)	coupled nonlinear	3rd roots	train(24^3)	1.28M	150	1.62E-1	**94.1**	**94.1**	**99.4**
	PDE's	linear	test(12^3)	127K	146	1.57E-1			
Swim	finite difference	const	ref(512^2)	3.68M	33.1	4.00E-1	**94.0**	**94.0**	**99.8**
(Spec95)	approximations for	2nd root	400^2	2.26M	33.0	4.00E-1	**98.7**	**98.7**	**99.8**
	shallow water equation	linear	200^2	568K	32.8	3.99E-1			
SP	computational fluid	const	50^3	4.80M	132	1.05E-1	**90.3**	**90.3**	**99.9**
(NAS)	dynamics (CFD)	3rd roots	32^3	1.26M	124	1.01E-1	**95.8**	**95.8**	**99.9**
	simulation	linear	28^3	850K	125	9.78E-2			
Tomcatv	vectorized mesh	const	ref(513^2)	1.83M	208	1.71E-1	**92.4**	**92.4**	**99.5**
(Spec95)	generation	2nd root	400^2	1.12M	104	1.67E-1	**77.3**	**99.2**	**99.3**
		linear	train(257^2)	460K	104	1.67E-1			
Hydro2d	hydrodynamic	const	ref	1.10M	13.4K	2.23E-1	**98.5**	**98.5**	**100**
(Spec95)	equations computing		train	1.10M	1.35K	2.23E-1	**98.5**	**98.4**	**100**
	galactic jets		test	1.10M	139	2.20E-1			
FFT	fast Fourier	const	512^2	1.05M	63.7	7.34E-2	**72.6**	**72.8**	**99.6**
	transformation	2nd root	256^2	263K	57.5	8.13E-2	**95.5**	**95.5**	**99.5**
		linear	128^2	65.8K	51.4	9.04E-2			
Mgrid	multigrid solver	const	ref(64^3)	956K	35.6K	6.81E-2	**96.4**	**96.4**	**100**
(Spec95)	in 3D potential	3rd roots	test(64^3)	956K	1.42K	6.76E-2	**96.5**	**96.5**	**99.3**
	field	linear	train(32^3)	132K	32.4K	7.15E-2			
Apsi	pollutant distribution	const	128x1x128	25.0M	6.35	1.60E-3	**27.2**	**91.6**	**97.8**
(Spec2K)	for	3rd roots	train(128x1x64)	25.0M	146	2.86E-4	**27.8**	**92.5**	**99.1**
	weather predication	linear	test(128x1x32)	25.0M	73.6	1.65E-4			

Table III. Prediction Accuracy and Coverage for Six Integer Programs

Benchmark	Description	Patterns	Inputs	Num. data elem.	Avg. reuses per elem.	Avg. dist. per elem.	Accuracy w/ data size (%)	Accuracy w/ sample size (%)	Cover-age (%)
Compress (Spec95)	an in-memory version of the common UNIX compression utility	const	ref	36.1M	628	4.06E-2	**86.1**	**85.9**	**92.2**
		linear	train	279K	314	6.31E-2	**92.3**	**92.3**	**86.9**
			test	142K	147	9.73E-2			
Twolf (Spec2K)	circuit placement and global routing, using simulated annealing	const	ref(1888-cell)	734K	177K	2.08E-2	**92.6**	**94.2**	**100**
		linear	train(752-cell)	402K	111K	1.82E-2	**96.2**	**96.6**	**100**
			370-cell	227K	8.41K	1.87E-2			
Vortex (Spec95) (Spec2K)	an object oriented database	const	ref	7.78M	4.60K	4.31E-4	**95.1**	**95.1**	**100**
			test	2.58M	530	3.25E-4	**97.2**	**97.2**	**100**
			train	501K	71.3K	4.51E-4			
GCC (Spec95)	based on the GNU C compiler version 2.5.3	const	expr	711K	137	2.75E-3	**98.2**	**98.2**	**100**
			cp-decl	705K	190	2.65E-3	**98.6**	**98.6**	**100**
			explow	321K	68.3	3.69E-3	**96.1**	**96.1**	**100**
			train(amptjp)	467K	221	3.08E-3	**98.7**	**98.7**	**100**
			test(cccp)	456K	233	3.25E-3			
Li (Spec95)	Xlisp interpreter	const	ref	87.9K	328K	2.19E-2	**85.6**	**82.7**	**100**
		linear	train	44.2K	1.86K	3.11E-2	**85.8**	**86.0**	**100**
			test	14.5K	37.0K	2.56E-2			
Go (Spec95)	an internationally ranked go-playing program	const	ref	109K	124K	3.78E-3	**96.5**	**96.5**	**100**
			test	104K	64.6K	3.78E-3	**96.9**	**96.9**	**100**
			train	86.1K	2.68K	2.02E-3			
						average	**88.6**	**93.5**	**99.1**

5 million. The longest trace is generated by the third input of *Twolf* and has over 130 billion memory references. No two inputs are similar in data size or execution length. The maximal reuse distance is very close to the data size in all programs.

We use three different input sizes for all programs except for *GCC*. Based on the two smaller inputs, we predict the largest input. We call this forward prediction. The prediction also works backwards: based on the smallest and the largest inputs, we predict the middle one. Locality in all executions can be thus predicted by extrapolation and interpolation. The prediction accuracy is shown by the 8th and 9th columns. The former, marked "Accuracy w/ data size," gives the prediction accuracy when using the number of distinct data elements as the input size. The latter, marked "Accuracy w/ sample size," gives the accuracy when using distance-based sampling.

For most benchmarks, the two columns give comparable results, which indicates a proportional relation between the input size and the data size. One exception is *Apsi* in Table II. For different input parameters, the program initializes the same amount of data but uses different portions of the data in computation. The prediction accuracy is only 27% using the data size but over 91% using distance-based sampling. In general, prediction based on sampling yields a higher accuracy.

Both forward and backward predictions are fairly accurate. Backward prediction is generally better except for *Lucas*—because the largest input is three orders of magnitude larger than the medium-size input—and for *Li*—because only the constant pattern is considered by the prediction. Among all prediction results, the highest accuracy is 99.2% for the medium-size input of *Tomcatv*, and the lowest is 72.8% for the large-size input of *FFT*. The average accuracy is 93.5%.

The last column shows the prediction coverage. The coverage is 100% for programs with only constant patterns because they need no sampling. For the others, the coverage starts after the input size is found in the execution trace. Let T be the length of the execution trace, and P be the logical time of the discovery; the coverage is $1 - P/T$. For programs using a reduced number of iterations, T is scaled up to the length of the full execution. To be consistent with other SPEC programs, we let programs *SP* and *FFT* have the same number of iterations as *Tomcatv*. Data sampling uses the first peak of the first two data samples for all programs with non-constant patterns except for *Compress* and *Li*. *Compress* needs 12 data samples. It is predictable in this test because it repeats compression multiple times. The results from program phase analysis show that *Gzip*, which uses the same algorithm as *Compress*, has the same locality when compressing files of different sizes and content [Shen et al. 2007]. *Li* has random peaks that cannot be consistently sampled. We predict *Li* based only on the constant pattern. The average coverage across all programs is 99.1%.

The actual coverage is smaller because the instrumented program (for sampling) runs slower than the original program. Our fastest analyzer causes a slowdown of 20 to 100 times. In the worst case, we need a coverage of at least 99% to finish prediction before the end of the execution. Fortunately, the low coverage happens only in *Compress*. Without *Compress*, the average coverage

Table IV. Five Methods of Locality Prediction

Models	*Single*	*Single*	*Multiple*	*Multiple*	*Multiple*
histogram	reference	reference	distance	distance	reference
histogram x-axis	log-linear	log-linear	logarithmic	log-linear	log-linear
num. inputs	2	3+	3+	3+	3+
num. patterns per bin	1	1	2+	2+	2+

Table V. Comparison of the Accuracy of Five Prediction Methods

	Single Model		Multi-model, 3+ Inputs			
	ref. hist.	ref. hist.	dist. hist.			Total
Bench-mark	2 inputs	3+ inputs	logarithmic	log-linear	ref. hist.	Inputs
Applu	92.06	97.40	93.65	93.90	90.83	6
Swim	94.02	94.05	84.67	92.20	72.84	5
SP	90.34	96.69	94.20	94.37	90.02	5
FFT	72.82	93.30	93.22	93.34	95.26	3
Tomcatv	92.36	94.38	94.70	96.69	88.89	5
GCC	98.61	97.95	98.83	98.91	93.34	4
Avg.	**90.04**	**95.63**	**93.21**	**94.90**	**88.53**	**4.7**

is 99.73%, suggesting 73% time coverage in online prediction on average. Even without a fast sampler, the prediction is still useful for long running programs and programs with mainly constant patterns. Six programs (or 40% of the test suite) do not need sampling.

4.2.2 *Multimodel Prediction.* Multimodel prediction allows us to examine three aspects of locality prediction in more depth: the use of multimodel prediction and all three types of histograms, the effect of using more than two training inputs, and prediction using very small inputs.

We compare five prediction methods listed in Table IV. The first two are single-model prediction using two training inputs and more than two inputs. Both use reference histograms computed from log-linear distance histograms. The next three are multimodel prediction using all three types of histograms: the reference histogram and the distance histogram in logarithmic and log-linear scale. All methods use sampling to measure the input size.

We restrict our attention to six test programs in Table V, which have multiple inputs and have some significant errors in single-model prediction. The second and third columns show that regression analysis on multiple inputs improves prediction accuracy. The largest improvement is from 73% to 93% in the case of *FFT*. Across the six programs, the average accuracy is raised from 90.0% to 95.6%.

The multimodel prediction using the logarithmic histogram is the most efficient among all methods, using merely 20 or fewer bins. The average accuracy, 93%, is comparable to the accuracy obtained from much larger histograms. However, low accuracy may result because the logarithmic scale produces large ranges that may hide important details. In one execution of *Swim*, 12% of the reuse distances occupy a narrow range between 260 and 280. Our prediction method assumes a uniform distribution in each range, so the accuracy is only 85% with logarithmic ranges, compared with 92% with log-linear ranges.

Table VI. Accuracy for *SP* with Small-Size Inputs

Largest Training	Testing Size	Single-Model 2 Inputs	Single-model 3+ Inputs	Multi-model Log Scale	Multi-model Log-linear Scale
8^3	10^3	79.61	79.61	85.92	89.50
	12^3	79.72	75.93	79.35	82.84
	14^3	69.62	71.12	74.12	85.14
	28^3	64.38	68.03	76.46	80.30
10^3	12^3	91.25	87.09	84.58	90.44
	14^3	81.91	83.20	78.52	87.23
	16^3	77.28	77.64	76.01	84.61
16^3	28^3	75.93	74.11	77.86	83.50

Multimodel prediction with reference histograms has the lowest average accuracy among the five methods, although it makes the best prediction for one program, *FFT*.

4.2.3 *Prediction Using Small Inputs.* An important assumption in all our prediction methods is that the composition of patterns remains constant across all executions. The assumption appears operable as shown by the accurate prediction we have observed so far. If we decrease the input size, the effect of nonrecurrent parts in a program, for example the initialization before a loop, becomes significant. Next we pick one program, *SP*, artificially reduce the input size, and evaluate our four best performing predictors.

Table VI shows the prediction accuracy when the size of the largest training run is reduced to 1.6%, 3%, and 13% of the size used previously in Table II. The two multimodel methods are up to 16% more accurate than the two single-model methods. Multimodel prediction using log-linear distance histograms is the most accurate, with accuracy ranging from 80% to 90%. The average accuracy is 7% higher than the next best method. Multimodel prediction using the logarithmic scale histogram does not perform as well for small inputs.

The ability of multimodel prediction to use very small inputs is useful in two cases. First, the training time is proportional to the size of the training runs, so very small training runs lead to very fast locality analysis. Second, it is often unclear how to determine when an input size is large enough, so the prediction is more reliable if it can maintain good accuracy across most input sizes.

4.3 Understanding Whole-Program Locality

Locality is considered a fundamental concept in computing because to understand a computation we must understand its use of data. The predictive models described in this article provide a new definition of locality that is quantitative, verifiable, whole-program based and applicable to any sequential system. It opens new ways to examine the active data usage in complex computations. We discuss several unique features of this work that have significant implications in how a programmer can better understand this important yet often elusive concept.

4.3.1 *Quantifying Locality as Patterns of Change.* Profiling has long been used to find invariant program properties. Examples include most used variables and functions [Wall 1991] and recurring program path [Hsu et al. 2002]

for feedback-guided optimization, "hot" memory instruction streams for runtime optimization [Chilimbi 2001b], and function-level [KleinOsowski and Lilja 2002] and general statistics [Eeckhout et al. 2002] for workload characterization. The constant pattern in this article represents invariant locality. In 15 programs, 4 have only the constant pattern. Its presence in the other 11 programs ranges from 28% in *Apsi* to 84% in *Twolf*. The average is 55%. For locality analysis, however, the constant pattern may be the least important because reuse distances in a constant pattern are usually short and do not cause misses in a large cache.

To fully characterize locality, locality prediction extends the use of profiling analysis to capture behavioral variations between executions. The analysis is aided by the property that reuse distance measures the recurrence independent of the instruction and data addresses involved in the data access.

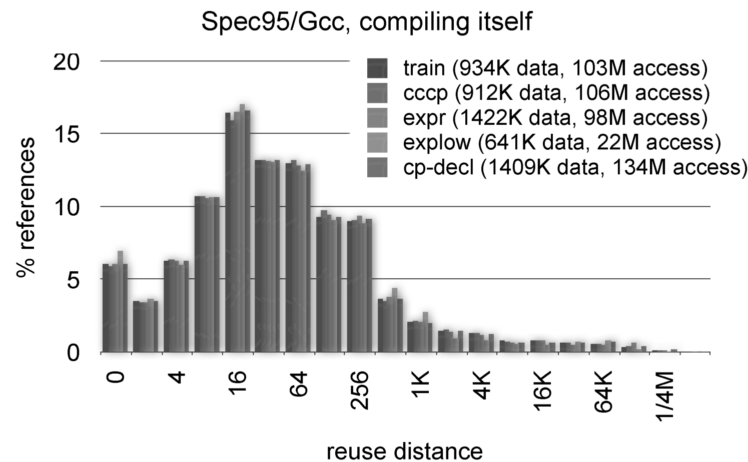
A careful reader may have noticed that for a number of programs, Tables II and III display a near identical number in the "average distance per element" column for different inputs. For example the number is 0.4 for all three inputs of *Swim*, which means that the average distance is 40% of the data size. The reason is that when the input size is sufficiently large, the total distance is dominated by reuse distances in the linear pattern. The same average distance is the result of the constant size of the linear pattern.

4.3.2 *Relation between a Program and Its Input.* The locality of the GNU C Compiler, *GCC*, is predicted with 96% to 99% accuracy, which is higher than people would normally expect. The program is large and complex—this version has 222,182 lines of source code in 120 files. More importantly, the data usage of the compiler should depend largely on the input. However, when measured using reuse distance, *GCC* manifests surprisingly regular locality. Figure 9 shows the locality signature of five executions of *GCC* when compiling some of its largest program files. The five signatures overlap by over 98%, which makes prediction accurate and trivial.

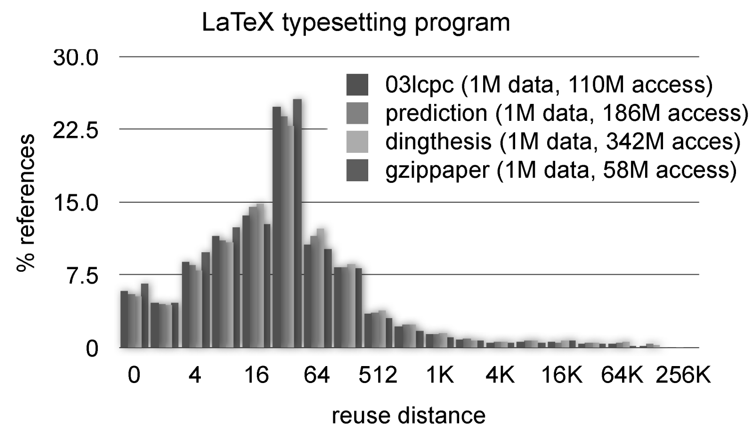
A compiler belongs to the general class of service-oriented programs. It processes input requests as a sequence of tasks, which in this case are the functions to be compiled. Shen et al. [2007] found that the compilation tasks go through the same sequence of phases in each task even though the length of the task is input dependent and unpredictable.

The input files we used for *GCC* have hundreds of functions. The signature might be showing the locality of compiling an "average" function. The consistency across inputs might be due to consistency in programmers' coding style, for example, the distribution of function sizes. To understand this more, we tested an extreme input, `166.i`, provided by the benchmark set. It contains two functions each with over one thousand lines of code. Distance-based sampling shows that long reuse distances are two orders of magnitude greater in *166.i* than in other inputs. The locality of compiling `166.i` is around 70% identical to the locality signatures shown in Figure 9(a), which suggests that the locality in *GCC* is 30% due to the input and 70% due to the program code.[1] This example

[1]Since the Alpha cluster has been replaced by a PC cluster at the time, we used the x86 binary instead of the Alpha binary.



(a) The locality signature of five executions of *GCC* compiling its largest program files



(b) The locality signature of four executions of the *LaTeX* typesetting program

Fig. 9. Locality signatures of *GCC* and *Latex*. The average locality is predictable even though the execution is largely input driven. The two programs have very similar signatures, which indicates common data usage patterns in compilation and type setting.

also suggests a general method—by testing a program on extreme inputs and measuring the deviance from the average—for exploring the range of variability in program locality.

4.3.3 *Relation between Programs.* With the whole-program locality model, we can now conduct comprehensive comparisons of the locality in different programs. In most cases, the locality signature is consistent in the same program but differs from one program to another. One interesting exception is the pair of programs *GCC* and *Latex*. *Latex* is a typesetting program commonly used in scientific publishing (including this article). Four locality signatures of

Latex, measured by Cheng [Cheng and Ding 2005] and shown in Figure 9(b), are strikingly similar to the locality signatures of *GCC* shown in the same figure. Intuitively, the result is expected since *GCC* and *Latex* are both language processors, one for the C programming language and the other for a type-setting language. However, it is significant that an automatic method can identify and quantify this similarity using an objective metric.

The two programs were developed independently, *Latex* by Knuth, Lamport, and other people mostly in universities, and *GCC* by a large group of open-source developers. In the introduction we mentioned that reuse distance is independent of localized transformations of a program or its execution. Here is an demonstration that the metric captures inherent similarity between two programs that share no code and have completely different data structures and program organization.

There may be practical uses in comparing program locality. A customer who is serious about performance may use locality as a metric in evaluating software from different vendors. A software company may be interested in maintaining performance (not just correctness) in new versions its products. The comparison can reveal changes in locality in different software versions and confirm or repudiate locality as a factor in complex performance problems. We have tested two versions of *GCC*, one in SPEC 1995 and the other in SPEC 2000 benchmark set, and found that their locality signatures overlap by 89%.

4.4 Summary

Compared to precise algorithms, approximate algorithms have faster performance and the performance scales better with the size of data. Relative-precision approximation provides unprecedented scalability, maintaining a constant speed regardless of the size of data and the length of reuse distance. At 99% accuracy, it measures the length of reuse distance in trillions at the same speed as exact solutions measure the length in millions. The scale tree it uses is orders of magnitude more space efficient than precise representations. A user can further improve the measurement speed by specifying a lower precision.

Using two training inputs, single-model prediction is 94% accurate on average for 15 benchmark programs, and its accuracy can be improved by 6% using more training data. Distance-based sampling can detect the input size after seeing less than 1% of an execution. The log-linear histograms always provide the most precise information for locality prediction, either in single-model and multimodel prediction.

Multimodel prediction can use logarithmic distance histograms and consequently be an order of magnitude more space efficient. Space efficiency is necessary for fine-grained analysis such as analyzing individual program instruction or data. In addition, multimodel prediction is 90% accurate with one fiftieth of the training data. However, the two efficiency boosts—logarithmic histograms and small inputs—should not be used at the same time.

Whole-program locality shows aggregate data usage in general sequential code. Our results show that a high degree of regularity in locality is a common phenomenon of collective behavior in complex code. The measurement and

prediction techniques are useful in observing and understanding these emergent effects, for example in understanding the relationships both between program code and input and between different programs.

5. RELATED WORK

This section discusses related work in program locality analysis. See Section 2.3 for related work in reuse distance measurement.

Training-Based Analysis. Independent of our work, Marin and Mellor-Crummey [2004, 2005] solved the same problem in the context of building a performance modeling toolkit. It uses compiler analysis to identify groups of related memory references for reuse-distance profiling. Given two histograms, their method first finds leading bins that have identical reuse distances and classifies them as constant patterns. Then it recursively divides the remaining group by its average reuse distance until the two halves show the same pattern of change in the two histograms, measured by the ratio between the average reuse distances. It uses quadratic programming to determine the best pattern parameters (because they model both locality and computation). A pattern function is a linear combination of base functions, which can include user supplied formula.

Recursive partitioning produces the minimal number of patterns with no loss of accuracy. By analyzing one reference group at a time, it can identify individual patterns that are difficult to separate in whole-program analysis. Recursive partitioning, however, is costly in data collection because it needs to store all reuse distances. In comparison, histograms with fixed-size bins are much more efficient to store, but they lose information about the distribution of reuse distances inside a bin. Multimodel prediction alleviates the problem by statistically estimating the mixing of patterns in the same bin.

Fang et al. [2005] solved the problem of measuring and analyzing the locality of every memory operation in a program. They used log-linear bins in data collection and experimented with different assumptions about the distribution within a bin. They found that a linear distribution worked well for both floating-point and integer programs while a uniform distribution (which we use in this work) worked well only for floating-point code. To improve efficiency, their method merges bins that have a similar distribution of reuse distances. The adaptive merging has a similar effect to Marin and Mellor-Crummey's recursive partitioning and makes the model compact without sacrificing its precision.

A common assumption in locality prediction is that a fixed fraction of reuse distances belong to each pattern in every locality signature. Marin and Mellor-Crummey [2004] tested heuristics not limited to this assumption but did not find them as stable and accurate. Our results in Section 4.2.2 showed that the assumption is mostly valid even for very small training inputs.

Static Analysis. Cascaval and Padua [2003] extended dependence analysis to estimate the reuse distances and the locality signature in scientific programs. Beyls and D'Hollander [2005] developed *reuse distance equations*, which

precisely compute the reuse distance in polyhedral loops using the Omega library [Kelly et al. 1996] as a fast symbolic but worst-case exponential-time solver. There are a number of compiler techniques that estimate the miss rate of a program [Porterfield 1989; Ferrante et al. 1991; Ghosh et al. 1999; Chatterjee et al. 2001; Xue and Vera 2004]. Compared to the miss rate, reuse distance is machine independent. Furthermore, it can be used to derive the miss rate. Beyls and D'Hollander [2005] compared compiler analysis with profiling by testing the effect of their use in cache-hint insertion.

A basic task of dependence checking is to analyze repeated accesses to data [Allen and Kennedy 2001; Wolfe 1996; Banerjee 1988]. Large-scale data usage can be summarized by various types of array section analysis [Havlak and Kennedy 1991], including linearization for high-dimensional arrays [Burke and Cytron 1986], linear inequalities for convex sections [Triolet et al. 1986], regular array sections [Callahan et al. 1988a], and reference lists [Li et al. 1990]. Other locality analyses include the matrix model [Wolf and Lam 1991; Kandemir 2005], memory ordering [McKinley et al. 1996], a number of later studies using high-dimensional discrete optimization [Cierniak and Li 1995; Kodukula et al. 1997], transitive closures [Song and Li 1999; Wonnacott 2002; Yi et al. 2000], and integer equations [Adve and Mellor-Crummey 1998].

Locality affects the fundamental balance between computation and memory transfer. Callahan et al. [1988b] defined the concept of program balance and machine balance. Techniques for matching the two balances have benefits from improving memory performance on conventional systems [Carr and Kennedy 1994; Ding and Kennedy 2004] to accelerating the design-space exploration in hardware-software co-design [So et al. 2002]. Whole-program locality can be used to estimate program balance in general-purpose applications.

Pure program analysis has a limited effect on general-purpose code because of the difficulty in analyzing complex control flow and indirect data accesses and characterizing aggregate program behavior. However, for regular loop nests with linearly indexed array references, static analysis can precisely model the iteration space and the data space. It can analyze locality in high-dimensional data, which is difficult for training-based analysis. In addition, compiler analysis is sound in that the result can be used to reorder program execution, for example, to change the program balance. Locality prediction is probabilistic. It measures common behavior but not all behavior. It cannot observe program behavior that does not occur in training runs. One solution is to combine compiler and profiling analysis, as demonstrated by Marin and Mellor-Crummey [2005]. Another solution is speculative program optimization. For example, Kelsey et al. [2009] proposed a software system that creates a FastTrack, which is a copy of a program optimized for the common behavior. The original code is run in parallel as a fallback in case the FastTrack code produces incorrect results.

Reuse Frequency and Data Streams. Access frequency gives the first model of data usage [Knuth 1971; Cocke and Kennedy 1974]. Later refinements include the lifetime of single objects [Seidl and Zorn 1998] or the affinity between data pairs [Thabit 1981; Calder et al. 1998; Chilimbi et al. 1999]. Chilimbi and

his colleagues extended the notion of affinity using *hot data streams*, which are repeated sequences of data accesses up to 100 elements long [Chilimbi 2001a, 2001b]. Streams and distances are complementary concepts. Hot streams show frequent repetitions, while the locality signature shows common recurrences. A stream contains order information that can be used for data prefetching. The locality signature provides a way to relate data by their aggregate locality (as used in reference affinity analysis described in Section 6).

Runtime Analysis. Saltz and his colleagues pioneered dynamic parallelization with the approach known as inspector-executor, where the inspector examines and partitions the data and computation at run time [Das et al. 1994]. Similar strategies were used to improve dynamic program locality [Ding and Kennedy 1999; Mellor-Crummey et al. 2001; Strout et al. 2003; Han and Tseng 2006]. Knobe and Sarkar [1998] included runtime data analysis in array static-single assignment (SSA) form. To reduce the overhead of runtime analysis, Arnold and Ryder [2001] developed a software framework for periodic sampling, which Chilimbi and Hirzel [2002] extended to discover hot data streams for data prefetching. Liu et al. [2004] developed a dynamic optimization system by leveraging hardware monitoring support for very low-cost sampling. In addition to sampling based on program code and hardware events, Ding and Kennedy [1999] sampled accesses to a subset of data in an array. Ding and Zhong [2002] extended the scheme for use on dynamic data. Distance-based sampling is a form of data-based sampling as it uses the reuse distance to select data samples.

While runtime analysis can identify patterns unique to the current execution, it is not as thorough as off-line training analysis. On the other hand, offline models and online analysis can be combined to help each other, as this article has shown in combining training analysis and distance-based sampling.

6. FIVE DIMENSIONS OF LOCALITY

Reuse distance has uses in numerous studies. As an imprecise and incomplete count, a keyword search in the ACM Digital Library shows 91 publications since 2003 that contain the phrase “reuse distance” in addition to the words “locality” and “cache” in conferences and journals in the area of programming systems, computer architecture, operating systems, and embedded systems. In this section, we present a taxonomy that classifies representative problems, solutions and uses of locality analysis in five mostly orthogonal dimensions: input, code, data, time, and execution environment.

Whole-Program Locality. Locality affected by the program input. The simple-model prediction described in this article has been used to predict the capacity miss rate for a set of programs across cache sizes, program inputs [Zhong et al. 2007], and program phases [Shen et al. 2004b]. Fang et al. [2005] extended whole-program prediction to predict the locality of each program instruction as a function of the input size. They defined a general concept called memory distance to include reuse, access, and value distance. Marin and Mellor-Crummey [2004] considered cache associativity and computational

characteristics and predicted performance across machine platforms. On parallel systems, the scalability of a program depends on the ratio of computation to communication. Locality determines the amount of communication. Rothberg et al. [1993] used simulation and curve fitting to derive the program locality for a SPLASH benchmark program, *Barnes-Hut*, which was too difficult for symbolic analysis.

Locality in Program Code. Beyls and D'Hollander [2002, 2005] used the locality signature of each instruction to generate *cache hints*, which guide cache replacement decisions in hardware so that the data loaded by low-locality instructions do not evict the data loaded by high-locality instructions. They reported performance improvements for both integer and floating-point benchmarks on Intel Itanium, demonstrating the first distance-based technique to directly improve performance. In a program, the locality of statements, loops, and functions can be analyzed using training analysis [Fang et al. 2005], compiler analysis (for scientific code) [Cascaval and Padua 2003; Beyls and D'Hollander 2005], or their combination [Marin and Mellor-Crummey 2004]. Beyls and D'Hollander [2005] compared profiling analysis and compiler analysis (called reuse distance equations) in generating cache hints.

Beyls and D'Hollander [2006a, 2006b] developed a program tuning tool *SLO*, which identifies the cause of long distance reuses and gives improvement suggestions for restructing the code. Using the tool, they were able to double the average speed of five SPEC 2000 benchmarks on four machine platforms. Furthermore, they used sampling analysis to reduce the profiling overhead from a 1000 times slowdown to a 5 times slowdown.

Locality in Program Data. As an optimization problem, data placement is theoretically intractable in general [Petrank and Rawitz 2002]. In practice, a useful metric is reference affinity, which identifies data that are used together. Zhong et al. [2004] defined reference affinity using reuse distance and showed its use in array regrouping and structure splitting. Zhang et al. [2006] showed formally that reference affinity uncovers the hierarchical locality in data from the access pattern in computation. Shen et al. [2005] developed a static analysis of reference affinity and tested its use in the IBM compiler. Zhao et al. [2007] included affinity in the Forma framework for automatic array reshaping in the IBM compiler.

Spatial locality measures the quality of a data layout. Three studies have defined spatial locality as the change in locality when the granularity of data increases from data elements to cache blocks [Berg and Hagersten 2005; Gu et al. 2009] or from data elements to memory pages [Bunt and Murphy 1984]. Gu et al. [2009] ranked program functions by (the lack of) spatial locality to aid program tuning.

Locality over Time. Batson and Madison [1976] defined a phase as a period of execution accessing a subset of program data. Denning [1980] stated that a proper model must account for the tranquility of phases as well as disruptive transitions. Shen et al. [2004a, 2007] built a model by effectively converting an execution to a signal, that is, a sequence of reuse distances, and applying

wavelet analysis to separate gradual changes from disruptive ones. Many subsequent studies considered wavelets and locality phases in modeling temporal behavior in a system.

For cache and memory management, a basic problem is predicting the time of future data access. Increasing evidence shows that the last reuse distance is an effective predictor. It has been used in memory management [Smaragdakis et al. 2003; Chen et al. 2005; Zhou et al. 2004] and file caching [Zhou et al. 2001; Jiang and Zhang 2002]. Kelly et al. [2004] found reuse distance to be a powerful predictor of response time in server systems. Almeida et al. [1996] showed that the locality signature of web reference streams follows log-normal distributions, in which the logarithm is normally distributed.

Interaction between Programs. When multiple running programs share a cache, the performance of one program is influenced by the locality of others. The effect can be modeled by inflating the reuse distance of the program with the footprint of its peers [Suh et al. 2001; Chandra et al. 2005]. Jiang et al. [2008] formulated the problem of optimal coscheduling on shared cache. For memory sharing, Yang et al. [2006] studied cooperative interaction between heap management in the Java virtual machine and memory management in the operating system. The key metric is the LRU reference histogram, which is equivalent to the locality signature in this article.

7. CONCLUSIONS

Locality has become increasingly important in the design of algorithms, compilers, operating systems, and computer architectures. In this article we have presented training-based whole-program locality analysis, which consists of two approximate algorithms for measuring reuse distance and five prediction methods for modeling whole-program locality. The approximate algorithms are faster and more scalable than exact solutions while guaranteeing an absolute or relative precision. The precision and cost are adjustable. The asymptotic cost of the relative-precision algorithm is effectively linear in the length of the trace. The five prediction methods decompose reuse distances using either reference histograms or distance histograms in logarithmic or log-linear scales. Each locality component can have a single pattern or multiple patterns. For 15 floating-point and integer benchmark applications, single-model prediction using two inputs shows 94% accuracy and 99% coverage. The accuracy can be improved by using more inputs and multimodel prediction. The efficiency can be improved by using compact histograms or very small inputs.

Locality is a fundamental aspect of computation. The new locality models in this article are quantitative yet they are not tied to any specific machine and are unaffected by irrelevant aspects of program construction. The results show that through them locality can be quantified for complex applications. The decomposition of locality as done in this work is orthogonal to the traditional decomposition of program code and data and hence provides a new dimension in program analysis. It provides a systematic model of application data behavior and a quantitative basis for understanding and managing dynamic data usage at different levels of a computing system.

ACKNOWLEDGMENTS

The authors wish to thank Grant Farmer, Wei Jiang, Roland Cheng, and Matthew Nettleton for their help in implementation and testing; Trishul Chilimbi, Sandhya Dwarkadas, Steve Dropsho, Xiaoming Gu, Matthew Hertz, Mark Hill, Bryan Jacobs, Gabriel Marin, John Mellor-Crummey, Joseph Modayil, Mitsu Ogihara, Michael Scott, Zhenlin Wang, Xiaoya Xiang, Mihalis Yannakaki, the anonymous reviewers of this journal, the LCR'02 workshop, PLDI'03 and LACSI'03 conferences for their comments, corrections and suggestions on the work and its presentation. Our experiments mainly used machines at Rochester purchased by several NSF Infrastructure grants and equipment grants from DEC/Compaq/HP and Intel. John Mellor-Crummey and Rob Fowler provided access to Rice Alpha clusters.

REFERENCES

ADVE, V. AND MELLOR-CRUMMEY, J. 1998. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

ALLEN, R. AND KENNEDY, K. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers.

ALMASI, G., CASCAVAL, C., AND PADUA, D. 2002. Calculating stack distances efficiently. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*.

ALMEIDA, V., BESTAVROS, A., CROVELLA, M., AND DE OLIVEIRA, A. 1996. Characterizing reference locality in the WWW. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*. 92–103.

ALON, N., MATIAS, Y., AND SZEGEDY, M. 1996. The space complexity of approximating the frequency moments. In *Proceedings of the ACM Symposium on Theory of Computing*.

ARNOLD, M. AND RYDER, B. G. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

BANERJEE, U. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA.

BATSON, A. P. AND MADISON, A. W. 1976. Measurements of major locality phases in symbolic reference strings. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*.

BENNETT, B. T. AND KRUSKAL, V. J. 1975. LRU stack processing. *IBM J. Resear. Devel.* 353–357.

BERG, E. AND HAGERSTEN, E. 2004. Statcache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 20–27.

BERG, E. AND HAGERSTEN, E. 2005. Fast data-locality profiling of native execution. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 169–180.

BEYLS, K. AND D'HOLLANDER, E. 2002. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*. Paderborn, Germany.

BEYLS, K. AND D'HOLLANDER, E. 2005. Generating cache hints for improved program efficiency. *J. Syst. Archit. 51*, 4, 223–250.

BEYLS, K. AND D'HOLLANDER, E. 2006a. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of the High-Performance Computing and Communications Council*. Springer. Lecture Notes in Computer Science, vol. 4208. 220–229.

BEYLS, K. AND D'HOLLANDER, E. 2006b. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *Proceedings of the ACM Conference on Computing Frontiers*.

BUNT, R. B. AND MURPHY, J. M. 1984. Measurement of locality and the behaviour of programs. *Comput. J. 27*, 3, 238–245.

BURKE, M. AND CYTRON, R. 1986. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*.

CALDER, B., KRINTZ, C., JOHN, S., AND AUSTIN, T. 1998. Cache-conscious data placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*.

CALLAHAN, D., COCKE, J., AND KENNEDY, K. 1988a. Analysis of interprocedural side effects in a parallel programming environment. *J. Paral. Distrib. Comput. 5*, 5, 517–550.

CALLAHAN, D., COCKE, J., AND KENNEDY, K. 1988b. Estimating interlock and improving balance for pipelined machines. *J. Paral. Distrib. Comput. 5*, 4, 334–358.

CARR, S. AND KENNEDY, K. 1994. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst. 16*, 6, 1768–1810.

CASCAVAL, C. AND PADUA, D. A. 2003. Estimating cache misses and locality using stack distances. In *Proceedings of the International Conference on Supercomputing*. San Francisco, CA.

CHANDRA, D., GUO, F., KIM, S., AND SOLIHIN, Y. 2005. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 340–351.

CHATTERJEE, S., PARKER, E., HANLON, P. J., AND LEBECK, A. R. 2001. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

CHEN, F., JIANG, S., AND ZHANG, X. 2005. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the USENIX Annual Technical Conference*.

CHENG, R. AND DING, C. 2005. Measuring temporal locality variation across program inputs. Tech. rep. TR 875, Department of Computer Science, University of Rochester.

CHILIMBI, T. M. 2001a. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

CHILIMBI, T. M. 2001b. On the stability of temporal data reference profiles. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

CHILIMBI, T. M., HILL, M. D., AND LARUS, J. R. 1999. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

CHILIMBI, T. M. AND HIRZEL, M. 2002. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

CIERNIAK, M. AND LI, W. 1995. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

COCKE, J. AND KENNEDY, K. 1974. Profitability computations on program flow graphs. Tech. rep. RC 5123, IBM.

DAS, R., UYSAL, M., SALTZ, J., AND HWANG, Y.-S. 1994. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Paral. Distrib. Comput. 22*, 3, 462–479.

DATAR, M., GIONIS, A., INDYK, P., AND MOTWANI, R. 2002. Maintaining stream statistics over sliding windows. *SIAM J. Comput. 31*, 6, 1794–1813.

DENNING, P. 1980. Working sets past and present. *IEEE Trans. Softw. Engin. 6*, 1.

DING, C. AND KENNEDY, K. 1999. Improving cache performance in dynamic applications through data and computation reorganization at runtime. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

DING, C. AND KENNEDY, K. 2004. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Paral. Distrib. Comput. 64*, 1, 108–134.

DING, C. AND ZHONG, Y. 2002. Compiler-directed runtime monitoring of program data access. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*.

EECKHOUT, L., VANDIERENDONCK, H., AND BOSSCHERE, K. D. 2002. Workload design: Selecting representative program-input pairs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*.

FANG, C., CARR, S., ONDER, S., AND WANG, Z. 2005. Instruction-based memory distance analysis and its application to optimization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*.

FERRANTE, J., SARKAR, V., AND THRASH, W. 1991. On estimating and enhancing cache effectiveness. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer-Verlag.

FLAJOLET, P. AND MARTIN, G. 1983. Probabilistic counting. In *Proceedings of the Symposium on Foundations of Computer Science*.

GHOSH, S., MARTONOSI, M., AND MALIK, S. 1999. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst. 21*, 4.

GU, X., CHRISTOPHER, I., BAI, T., ZHANG, C., AND DING, C. 2009. A component model of spatial locality. In *Proceedings of the International Symposium on Memory Management*.

HAN, H. AND TSENG, C.-W. 2006. Exploiting locality for irregular scientific codes. *IEEE Trans. Paral. Distrib. Syst. 17*, 7, 606–618.

HAVLAK, P. AND KENNEDY, K. 1991. An implementation of interprocedural bounded regular section analysis. *IEEE Trans. Paral. Distrib. Syst. 2*, 3, 350–360.

HILL, M. D. AND SMITH, A. J. 1989. Evaluating associativity in CPU caches. *IEEE Trans. Comput. 38*, 12, 1612–1630.

HSU, W., CHEN, H., YEW, P. C., AND CHEN, D. 2002. On the predictability of program behavior using different input data sets. In *Proceedings of the 6th Workshop on Interaction Between Compilers and Computer Architectures (INTERACT)*.

JIANG, S. AND ZHANG, X. 2002. LIRS: An efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*.

JIANG, Y., SHEN, X., CHEN, J., AND TRIPATHI, R. 2008. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 220–229.

KANDEMIR, M. T. 2005. Improving whole-program locality using intra-procedural and inter-procedural transformations. *J. Paral. Distrib. Comput. 65*, 5, 564–582.

KELLY, T., COHEN, I., GOLDSZMIDT, M., AND KEETON, K. 2004. Inducing models of black-box storage arrays. Tech. rep. HPL-2004-108, HP Laboratories Palo Alto, CA.

KELLY, W., MASLOV, V., PUGH, W., ROSSER, E., SHPEISMAN, T., AND WONNACOTT, D. 1996. The Omega Library Interface Guide. Tech. rep., Department of Computer Science, University of Maryland, College Park.

KELSEY, K., BAI, T., AND DING, C. 2009. Fast track: A software system for speculative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*.

KIM, Y. H., HILL, M. D., AND WOOD, D. A. 1991. Implementing stack simulation for highly-associative memories. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 212–213.

KLEINOSOWSKI, A. AND LILJA, D. J. 2002. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Comput. Archit. Lett. 1*.

KNOBE, K. AND SARKAR, V. 1998. Array SSA form and its use in parallelization. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

KNUTH, D. 1971. An empirical study of FORTRAN programs. *Softw. Pract. Exper. 1*, 105–133.

KODUKULA, I., AHMED, N., AND PINGALI, K. 1997. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

LI, Z., YEW, P., AND ZHU, C. 1990. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. Paral. Distrib. Syst. 1*, 1, 26–34.

LIU, J., CHEN, H., YEW, P.-C., AND HSU, W.-C. 2004. Design and implementation of a lightweight dynamic optimization system. *J. Instruct.-Level Paral. 6*.

MARIN, G. AND MELLOR-CRUMMEY, J. 2004. Cross architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*.

MARIN, G. AND MELLOR-CRUMMEY, J. 2005. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Las Alamos Computer Science Institute*.

MATTSON, R. L., GECSEI, J., SLUTZ, D., AND TRAIGER, I. L. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9*, 2, 78–117.

MCKINLEY, K. S., CARR, S., AND TSENG, C.-W. 1996. Improving data locality with loop transformations. *ACM Trans. Program. Lang. Syst. 18*, 4, 424–453.

MELLOR-CRUMMEY, J., WHALLEY, D., AND KENNEDY, K. 2001. Improving memory hierarchy performance for irregular applications. *Int. J. Paral. Program. 29*, 3.

OLKEN, F. 1981. Efficient methods for calculating the success function of fixed space replacement policies. Tech. rep. LBL-12370, Lawrence Berkeley Laboratory.

PETRANK, E. AND RAWITZ, D. 2002. The hardness of cache conscious data placement. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

PORTERFIELD, A. 1989. Software methods for improvement of cache performance. Ph.D. thesis, Department of Computer Science, Rice University.

RAWLINGS, J. O. 1988. *Applied Regression Analysis: A Research Tool*. Wadsworth and Brooks.

ROTHBERG, E., SINGH, J. P., AND GUPTA, A. 1993. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*. 14–25.

SEIDL, M. L. AND ZORN, B. G. 1998. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*.

SHEN, X., GAO, Y., DING, C., AND ARCHAMBAULT, R. 2005. Lightweight reference affinity analysis. In *Proceedings of the 19th ACM International Conference on Super-Computing*. 131–140.

SHEN, X., SHAW, J., MEEKER, B., AND DING, C. 2007. Locality approximation using time. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 55–61.

SHEN, X., ZHANG, C., DING, C., SCOTT, M., DWARKADAS, S., AND OGIHARA, M. 2007. Analysis of input-dependent program behavior using active profiling. In *Proceedings of The 1st Workshop on Experimental Computer Science*.

SHEN, X., ZHONG, Y., AND DING, C. 2004a. Locality phase prediction. In *Proceedings of the International Conference on Architectual Support for Programming Languages and Operating Systems*. 165–176.

SHEN, X., ZHONG, Y., AND DING, C. 2004b. Phase-based miss rate prediction. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*.

SHEN, X., ZHONG, Y., AND DING, C. 2007. Predicting locality phases for dynamic memory optimization. *J. Paral. Distrib. Comput. 67*, 7, 783–796.

SLEATOR, D. D. AND TARJAN, R. E. 1985. Self adjusting binary search trees. *J. ACM 32*, 3.

SMARAGDAKIS, Y., KAPLAN, S., AND WILSON, P. 2003. The EELRU adaptive replacement algorithm. *Perform. Eval. 53*, 2, 93–123.

SMITH, A. J. 1976. On the effectiveness of set associative page mapping and its applications in main memory management. In *Proceedings of the 2nd International Conference on Software Engineering*.

SO, B., HALL, M. W., AND DINIZ, P. C. 2002. A compiler approach to fast hardware design space exploration in FPGA-based systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

SONG, Y. AND LI, Z. 1999. New tiling techniques to improve cache temporal locality. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

SRIVASTAVA, A. AND EUSTACE, A. 1994. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

STROUT, M. M., CARTER, L., AND FERRANTE, J. 2003. Compile-time composition of runtime data and iteration reorderings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–257.

SUGUMAR, R. A. AND ABRAHAM, S. G. 1993. Multi-configuration simulation algorithms for the evaluation of computer architecture designs. Tech. rep., University of Michigan.

SUH, G. E., DEVADAS, S., AND RUDOLPH, L. 2001. Analytical cache models with applications to cache partitioning. In *Proceedings of the International Conference on Super-Computing*. 1–12.

THABIT, K. O. 1981. Cache management by the compiler. Ph.D. thesis, Department of Computer Science, Rice University.

THOMPSON, J. G. AND SMITH, A. J. 1989. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Trans. Comput. Syst. 7*, 1, 78–117.

TRIOLET, R., IRIGOIN, F., AND FEAUTRIER, P. 1986. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*.

WALL, D. W. 1991. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

WANG, W. AND BAER, J.-L. 1991. Efficient trace-driven simulation methods for cache performance analysis. *ACM Trans. Comput. Syst. 9*, 3.

WOLF, M. E. AND LAM, M. 1991. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*.

WOLFE, M. J. 1996. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA.

WONNACOTT, D. 2002. Achieving scalable locality with time skewing. *Int. J. Paral. Program. 30*, 3.

XUE, J. AND VERA, X. 2004. Efficient and accurate analytical modeling of whole-program data cache behavior. *IEEE Trans. Comput. 53*, 5.

YANG, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. 2006. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 103–116.

YI, Q., ADVE, V., AND KENNEDY, K. 2000. Transforming loops to recursion for multi-level memory hierarchies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

ZHANG, C., DING, C., OGIHARA, M., ZHONG, Y., AND WU, Y. 2006. A hierarchical model of data locality. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

ZHAO, P., CUI, S., GAO, Y., SILVERA, R., AND AMARAL, J. N. 2007. Forma: A framework for safe automatic array reshaping. *ACM Trans. Program. Lang. Syst. 30*, 1, 2.

ZHONG, Y. AND CHANG, W. 2008. Sampling-based program locality approximation. In *Proceedings of the International Symposium on Memory Management*. 91–100.

ZHONG, Y., DING, C., AND KENNEDY, K. 2002. Reuse distance analysis for scientific programs. In *Proceedings of Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*.

ZHONG, Y., DROPSHO, S. G., SHEN, X., STUDER, A., AND DING, C. 2007. Miss rate prediction across program inputs and cache configurations. *IEEE Trans. Comput. 56*, 3, 328–343.

ZHONG, Y., ORLOVICH, M., SHEN, X., AND DING, C. 2004. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

ZHOU, P., PANDEY, V., SUNDARESAN, J., RAGHURAMAN, A., ZHOU, Y., AND KUMAR, S. 2004. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

ZHOU, Y., CHEN, P. M., AND LI, K. 2001. The multi-queue replacement algorithm for second-level buffer caches. In *Proceedings of the USENIX Technical Conference*.

Received September 2004; revised September 2005; accepted May 2007