Journal of Instruction-Level Parallelism 7 (2005) 1-28 Submitted 02/2005; published 06/2005

Understanding the Impact of Inter-Thread Cache Interference on ILP in Modern SMT Processors

Joshua Kihm KIHM@COLORADO.EDU
Alex Settle SETTLE@COLORADO.EDU
Andrew Janiszewski JANISZEW@COLORADO.EDU
Dan Connors DCONNORS@COLORADO.EDU
University of Colorado at Boulder
Department of Electrical and Computer Engineering
425 UCB, Boulder, Colorado

Abstract

Simultaneous Multithreading (SMT) has emerged as an effective method of increasing utilization of resources in modern super-scalar processors. SMT processors increase instruction-level parallelism (ILP) and resource utilization by simultaneously executing instructions from multiple independent threads. Although simultaneously sharing resources benefits system throughput, co-scheduled threads often aggressively compete for limited resources, namely the cache memory system. While compiler and hardware technologies have been traditionally examined for their effect on ILP, in the context of SMT machines, the operating system also has a substantial influence on system performance. By making informed scheduling decisions, the operating system can limit the amount of contention in the memory hierarchy between threads and reduce the impact of multiple threads simultaneously accessing the cache system. This paper explores the design of a novel fine-grained hardware cache monitoring system in an SMT-based processor that enables improved operating system scheduling and recaptures parallelism by mitigating interference.

1. Introduction

Simultaneous Multithreading (SMT) [1, 2, 3, 4] has emerged as a leading architecture model to achieve high performance in scientific and commercial workloads. By simultaneously executing instructions from multiple threads, SMT architectures maximize on-chip parallelism by converting independent thread parallelism to instruction-level parallelism (ILP) [5], thereby improving the utilization of architecture resources. During periods when individual threads might otherwise stall and reduce the efficiency of a large single-threaded processor, SMT improves execution throughput by executing instructions from other threads and thereby maintaining on-chip parallelism. Although an SMT machine can compensate for periods of low ILP due to cache miss delays and branch mispredictions by executing instructions from alternate applications, there are severe limitations to SMT designs. Among the most pressing of these is contention between threads for the limited capacity of the memory hierarchy.

Traditionally, researchers have explored microarchitecture ([6]) and compiler techniques ([7]) to maximize ILP in the presence of increasing cache miss penalties. On the other hand, in the context of SMT processors both the SMT microarchitecture and operating system can adapt policies to lessen the memory bottleneck. Microarchitecture ideas in SMT either have been intelligent out-of-order issue or in the area of adaptive cache management. For example, several fetch mechanisms for SMT are explored in [8] which were found to increase overall throughput dramatically. Different methods of dividing cache space have also been explored. Suh, et al. ([9]) explore methods for assigning ways within a cache set to each thread as to maximize throughput. Fairness is added to this scheme in [10]. Finally, reserving certain ways of the cache for a high priority thread was proposed in [11].

The operating system controls resources and activities at much larger time scales than hardware and can also greatly affect overall performance. Current operating system scheduling techniques and traditional hardware approaches have not been sufficiently exploited to improve the interaction of threads in the presence of the widening gap between processor and memory latencies . For instance, although current SMT techniques such as Intel's HyperThreading [12] show improvements in throughput, studies [13] show that contention for cache resources in SMT designs can have equally negative effects on application performance. Previous work ([14, 15]) has demonstrated the dramatic effect of OS scheduling decisions on multithreaded performance and propose techniques to make these decisions intelligently. These techniques are generally directed using prior knowledge of program execution behavior. Performance of thread combinations in previous scheduling periods, for example, can be used to predict the performance of those combinations in future scheduling periods. Even with profile-based scheduling decisions, however, such approaches have a limited ability to adapt to wide variations in performance of engineering and commercial workloads over time and their impact on modern memory systems.

In order to increase the cache system effectiveness for SMT architectures, this work investigates methods of *run-time guided thread management*, in which cache contention between threads is minimized by exposing novel architecture performance monitoring features to the operating system job scheduler. Our scheme seeks to aid thread scheduling by gathering run-time cache use and miss patterns for each active hardware thread at a fine granularity. Overall, we show that the use of fine-grained cache activity information in thread scheduling reduces inter-thread interference in the cache by approximately 10%, improving overall cache hit rates. This works to mitigate the cache bottleneck and clear the way for increased parallelism, as evidenced by an observed improvement in the performance of simultaneously multithreaded systems over existing scheduling techniques by roughly **5%**.

Although this paper focuses specifically on SMT, the techniques presented could be applied to any class of multithreading where multiple threads share cache space. These include many chip multiprocessors (CMP), fine-grained multithreading ([16]), and coarse-grained multithreading ([17, 18]). In this paper, SMT was chosen due to the readily available, popular commercial hardware (the Intel Pentium-4) which implements a form of SMT. This also allows monitoring interference at higher level caches which are not shared in some other multithreading paradigms and the ability to monitor the limitations on ILP imposed by cache interference.

The remainder of this paper is organized as follows. Section 2 provides an empirical analysis of thread conflicts and cache contention in SMT machines and the motivation behind our proposed approach. Next, Section 3 introduces our model and Section 4 presents an overview of our experimental methodology in characterizing and minimizing cache interference in simultaneous multithreaded architectures. The link between cache activity and interference is explored, and the observed effectiveness in improving performance and exploiting cache resource efficiency in an SMT architecture is presented in Section 5. Related work on multithreading and SMT optimization is discussed in Section 6. Finally, conclusions and future research areas are outlined in Section 7.

2. Motivation

2.1 Variation in Cache Activity

The demand placed on a cache level by a given program will vary both temporally and spatially. Cache activity can vary across time as data sets are traversed or because of changes in program phase. Additionally, activity in the cache is oftentimes much higher in certain regions than others. This phenomenon is illustrated in Figure 1. In the figure, the vertical axis represents position in the cache, which is broken up into 32 regions of 4 consecutive sets each, called a *super set*. The horizontal axis represents time, which is broken up to a granularity of five million executed cycles. The color of a point is determined by the number of accesses to that cache region during that sample, with bright colors (yellow) indicating high activity and dull colors (blue) indicating low activity. The cache model used in this experiment is a unified, 8-way associative, 512KB Level 2 cache with 128 byte blocks. Variation across the cache is illustrated by the variation in color along a vertical line. Program phase changes are illustrated by changes in the horizontal direction. The floating point benchmark *183.equake*, for example, goes through a warm-up phase for approximately 100 samples before entering a periodic cycle approximately seven samples long which represent iterations of its main calculation algorithm.

In many cases, the demand of a given thread is concentrated in a very small amount of the cache, as illustrated in Figure 2. The vertical axis is the percentage of samples exhibiting a particular behavior. The dark-colored, bottom sections of the bars indicate the percentage of samples with less than the global median amount of activity. These samples are relatively unimportant as they are unlikely to stress the memory system and they may give misleading results as to the concentration of cache accesses. If there are only a few cache accesses, having all of them occur in the same cache region is not a significant phenomenon. The second sections indicate samples where activity is fairly evenly spread across the cache. This is the typical case for the benchmark *181.mcf*. The interesting cases are the top regions. In these samples, 60% or more of the cache accesses occur in one half or one eighth of the cache respectively. These regions are referred to as 'hot' since they account for the majority of an application's cache accesses.

The hot regions explain the inconsistency in SMT performance across application sets. As stated in Section 1 some job mixes benefit considerably from SMT architectures, while others experience performance losses. Thus, for an SMT system to achieve the potential improvements in ILP provided by this architecture the machine must be able to prevent jobs with similar hot cache regions from interfering with each other. Since the operating system ultimately controls which jobs are co-scheduled on an SMT machine, it is a natural extension to provide the scheduler with low level cache information to prevent this type of interference from occurring. Thus, the challenge for the operating system is to co-schedule jobs that predominantly use different regions of the cache during a scheduling interval.

2.2 Periodicity and Predictability of Cache Activity

One of the inherent difficulties of scheduling to minimize interference is that decisions have to be made before the interval occurs. The scheduling decision is made before the interference occurs, so some level of prediction of thread behavior is required. A prediction may be as simple as assuming that past behavior will be perfectly indicative of future performance. Unfortunately, as programs change phase, so do their cache access patterns, which in turn affects interference [19], making

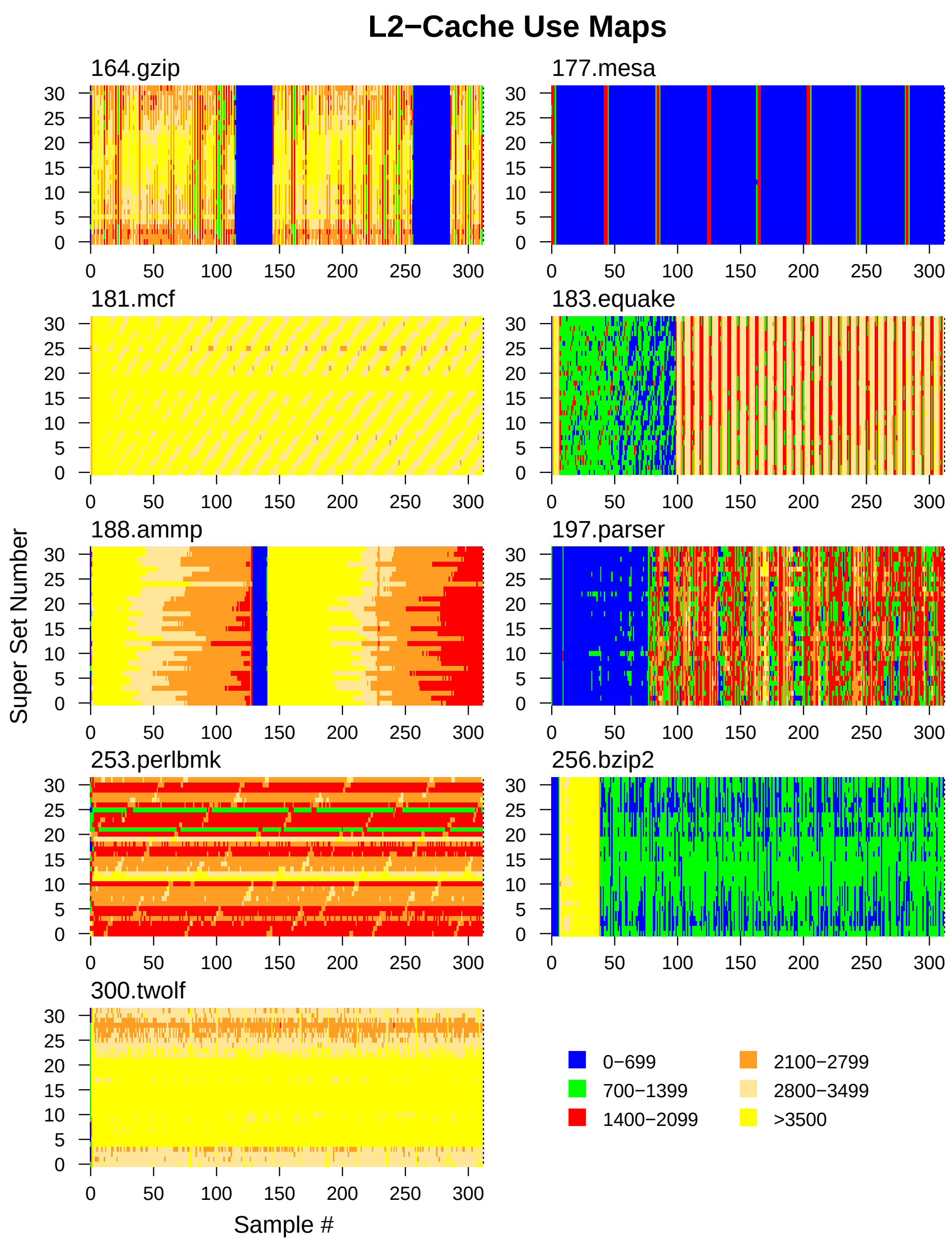


Figure 1: Unified Level-2 access patterns across cache region and time for nine *Spec2000* benchmarks.

this assumption unsafe. On the other hand, program phase is typically periodic, and hence easily predictable [20]. Fine grained cache behavior is also periodic and predictable.

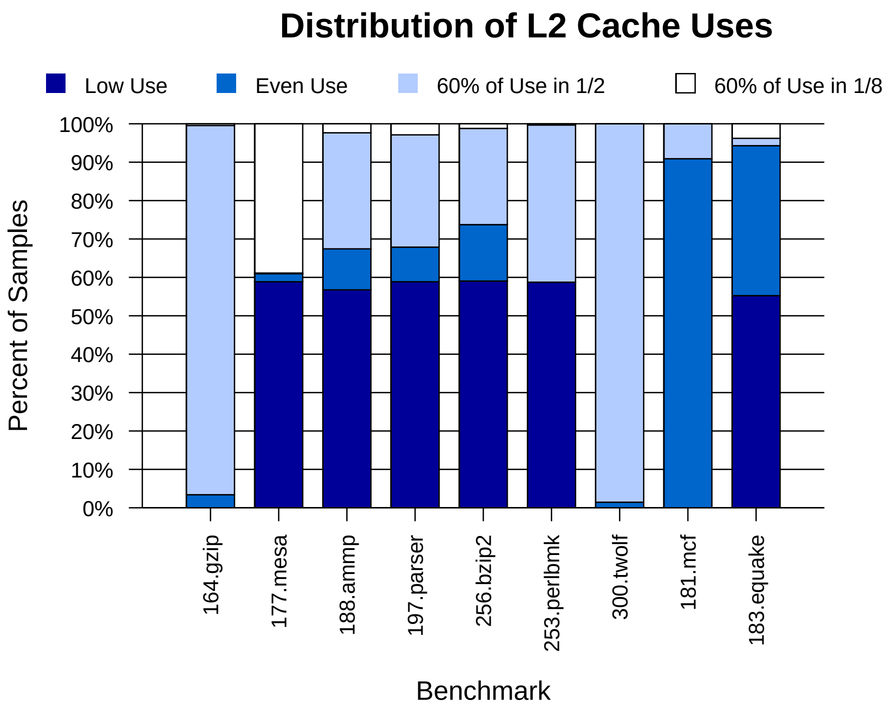


Figure 2: Concentration of Level 2 accesses for nine *Spec2000* benchmarks.

The periodicity of activity in certain benchmarks can be illustrated by performing a two-dimensional, discrete Fourier transform (DFT) ([21]) on the activity data. By transforming the data into the frequency domain, it is easier to recognize the periodic behaviors of the benchmarks. The combination of several components of different frequencies makes it difficult to identify each component in the time domain, but they are easily differentiated in the frequency domain. The Fourier transform of the data in Figure 1 is presented in Figure 3. The vertical axis now represents the frequency components of the activity across the cache, and the horizontal axis shows temporal frequency components. Warm colors indicate a large frequency component and cool colors indicate a weak component. In the DFT, frequency ranges from zero, the constant component, to one half, which represents switching back and forth between two values every sample. Additionally, the magnitude of the DFT of a real value signal is symmetric about zero. In the graphs, vertical lines indicate strong temporal periodicities, which is precisely what is needed to make good predictions. For example, the vertical bars at frequencies $\pm.4$ in the data for the compression benchmark *164.gzip* indicate that the activity is periodic approximately every 2.5 samples. In this case, having the history of the last two samples and the knowledge of periodicity would allow a good prediction of the next sample. Horizontal lines indicate periodicity in usage across the cache. The horizontal lines at approximately $\pm.33$ in *188.ammp* indicate that every third region is used approximately equally. A strong component at frequency zero, the constant value component, indicates the offset from zero, or the average of the function. Hence the benchmarks with heavy overall activity have large components around the horizontal and vertical axes. In [22], it is shown that low-order, auto-regressive models could be used to predict behavior in upcoming samples very accurately based on this periodicity. Using approximately five previous samples, it is possible to predict the use in a given cache region to within 5% root-mean-squared error. In other words, the periodic behaviors can be captured and used to make accurate predictions with a very short memory.

2.3 Inter-Thread Interference

A simple mechanism to measure the effectiveness of multithreading at creating ILP and the coinciding overall interference between threads is presented in [23]. The measure is to find the ratio between the IPC that a thread achieves in a multithreaded system and the IPC that it would achieve when run in the absence of other threads. These ratios are then added together to get a relative throughput of the system. The relative throughput of the threads is also a good indicator of fairness, with balanced throughput indicating each thread is sacrificing the same percentage of performance to inter-thread interference. The maximum relative throughput of a system is equal to the number of threads, indicating that each thread is running as fast as it would in the absence of the others. A relative throughput of one (or 100%) means that there is no gain from multithreading, and a relative throughput of less than one means that interference between the threads is greater than the benefit of multithreading. The advantage of using relative throughput is that it does not favor faster running threads as a simple IPC measure does. The number of *inter-thread kickouts* (ITKO) is one effective measure of inter-thread cache interference. An ITKO occurs when an access by one thread causes the eviction of a block of data from another. If the second thread tries to access this data again, it will experience a cache miss which it would not have experienced in a single-threaded environment.

In Figure 4, the relative throughput and total ITKO in the level-3 cache are shown versus time for a pairing of *181.mcf* and *183.equake*. The shaded regions indicate the relative throughput of each of the benchmarks, with the total indicating the overall relative throughput. The dark line indicates the total number of ITKO in the level-3 cache for each sample. During the brief initial period of approximately one million cycles, there is a large number of ITKO and very low throughput. This is followed by a longer period of high throughput and and low ITKO. Finally, as *equake* enters its main calculation iterations, periodic spikes in throughput coincide with dips in ITKO in a long period of low throughput and high interference. The correlation between ITKO and throughput in this data is further illustrated in Figure 5. In this graph, relative throughput is plotted as a function of ITKO. The correlation is weakened by the fact that many other factors that affect throughput such as interference in other levels of the cache or limited resources will also vary between samples. However, a decrease in ITKO generally corresponds to a noticeable increase in throughput. Clearly, methods that minimize ITKO will likely have a strong positive impact on throughput and therefore on IPC and ILP.

As the activity of each co-scheduled thread varies with time and across the cache, the interference between the threads will also vary. Shown in Figure 6 is a color map of the interference between static pairings of *Spec2000* benchmarks. These graphs are similar to Figure 1, except in this case the cache pictured is the level one data cache and the color is determined by the level of interference between the threads. The vertical axis indicates position in the cache, which is broken up into 32 contiguous regions for this experiment. The horizontal axis indicates execution time which is broken up into intervals of five million clock cycles. The color of the graph indicates how

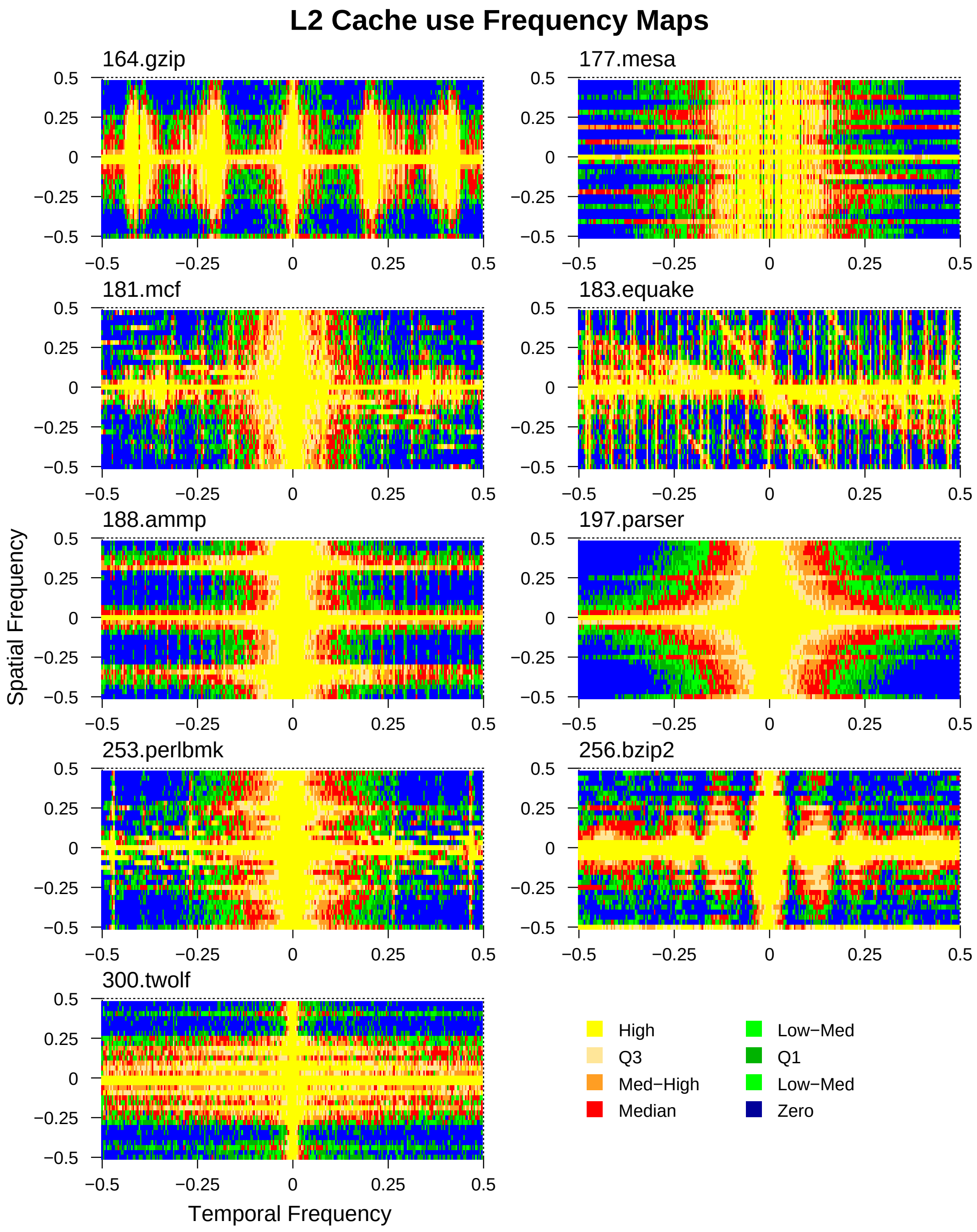


Figure 3: Frequency domain representation of unifi ed level 2 access patterns across cache region and time for nine *Spec2000* benchmarks.

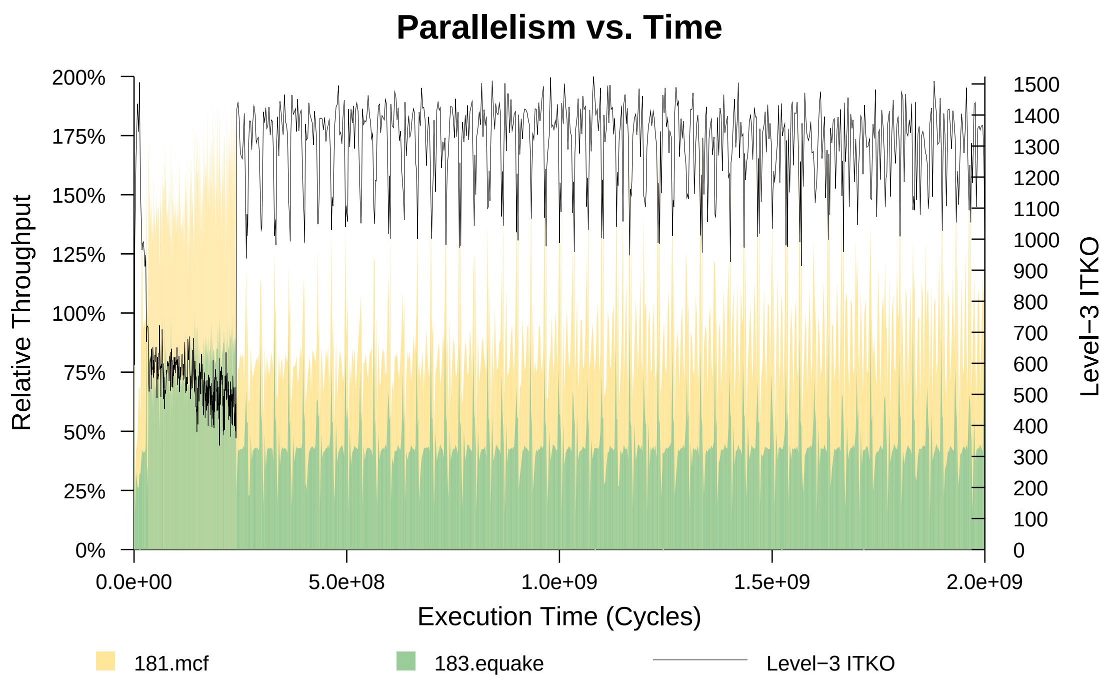


Figure 4: Relative throughput and Level-3 ITKO versus time for SMT pairing of *181.mcf* and *183.equake*.

many ITKO occur in that cache region during that sample. The brighter the color, the more ITKO. The cache model is an 8KB, four-way associative cache with 64 byte block size. The benchmarks are started together and allowed to run together for the duration of the test. The first 1.5 billion execution cycles are shown. The first important characteristic to notice is that the interference patterns vary greatly between pairings, so the choice of which threads to run together is vital in order to minimize interference. The next important observation is that interference varies over time. As the threads change phase, both their overall demand on the cache and where that demand is concentrated changes. As such, the interference between the two threads varies significantly. Figure 7 illustrates a simple experiment in dynamic scheduling. The representation is the same as in Figure 6, however, instead of allowing the threads to run together for the duration of the experiment, one of the threads is allowed to run continuously while the other context is filled with a thread which minimizes interference for the next sample chosen from the other three threads. This reduces overall interference by approximately 10% over the best static pairing and demonstrates that dynamic scheduling can avoid execution intervals where the phases of the constituent threads have the highest levels of interference and thereby outperform any static scheduling methodology.

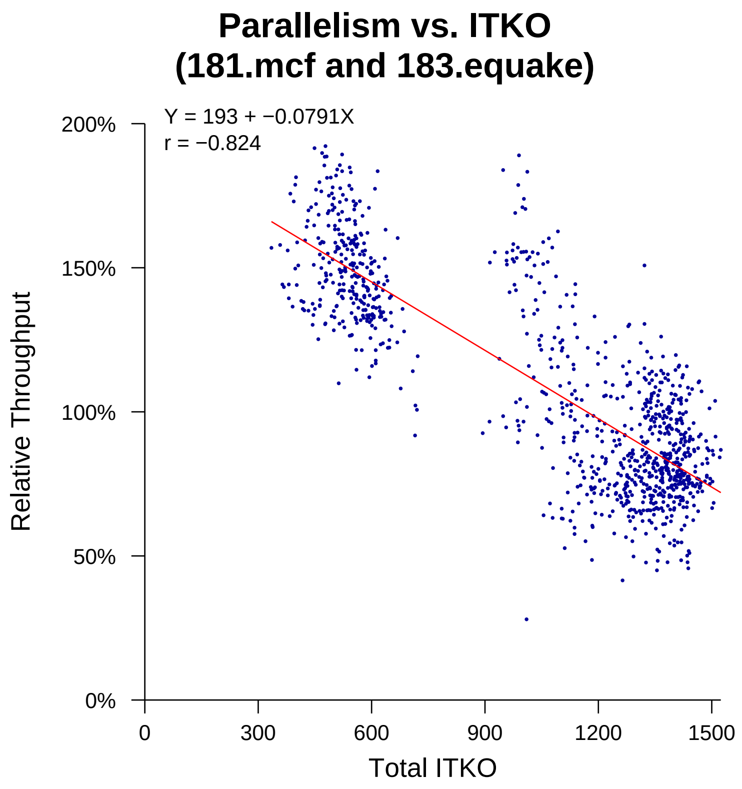


Figure 5: Relative throughput versus Level-3 ITKO for SMT pairing of *181.mcf* and *183.equake*. An increase in ITKO corresponds to a decrease in throughput

3. Theoretical Model

In multithreaded systems, opportunities to improve system behavior via adaptation occur at three time scales - the microarchitecture controls activities that occur in the tens to hundreds of cycles, the runtime system controls activities on the scale of thousands of cycles, and the operating system (OS) controls resources and activities at the largest time scales, millions of cycles. The OS impacts the performance most signifi cantly by making intelligent co-scheduling decisions which can be improved by leveraging the cache activity information of individual threads. By accurately estimating heavily accessed cache areas, the OS can co-schedule threads which will simultaneously use different cache regions, and thereby minimize interference. Only by minimizing interference between the threads in the cache system will the effi ciency of a multithreaded processor be maximized to achieve better throughput and higher ILP.

In order to take advantage of the variance in activity across a cache level, some mechanism must be developed to monitor the activity in different regions of the cache. The cache is fi rst divided into a series of regions called *super-sets*. A super-set is made up of a number of consecutive cache sets. All super-sets are the same size. If the number of sets per super-set is a power of two, the super-

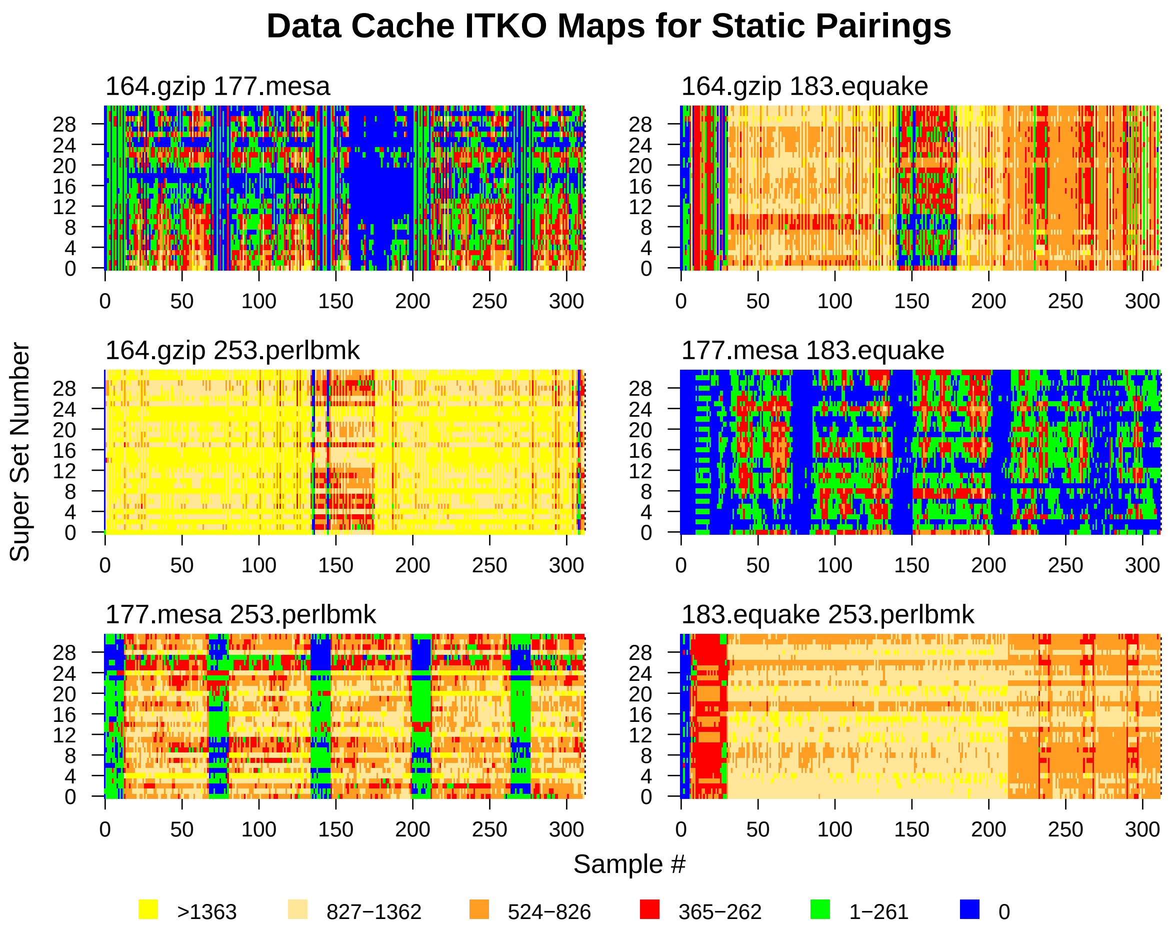


Figure 6: Level-1 d-cache inter-thread interference maps of benchmark pairings using static pairing.

set index is simply the high-order bits of the cache line index, making it very simple to monitor cache requests to determine the activity in each super-set. The number of accesses or misses in each super-set for each context is monitored by an *activity counter*. Due to the noise inherent in activity data and the cost of maintaining high precision, it is preferable to reduce these counters to a small number of bits. These reduced activity counters are then concatenated into an *activity vector*. An activity vector contains all of the data for a thread for the previous scheduling interval.

The hardware necessary to achieve this is illustrated in Figure 8. The first step is that the cache request bus is snooped such that the high-order index bits (the super-set number) and the thread ID bits are monitored for each cache request. Monitoring cache misses is simply a matter of snooping the cache requests to the next lower level of the cache hierarchy. These bits are used to index into a register file, the activity counters, which contain one register for each super-set and for each context, so that the total number of registers is the product of the number of contexts and the number of super-sets. The precision of the registers is dependent on the length of the

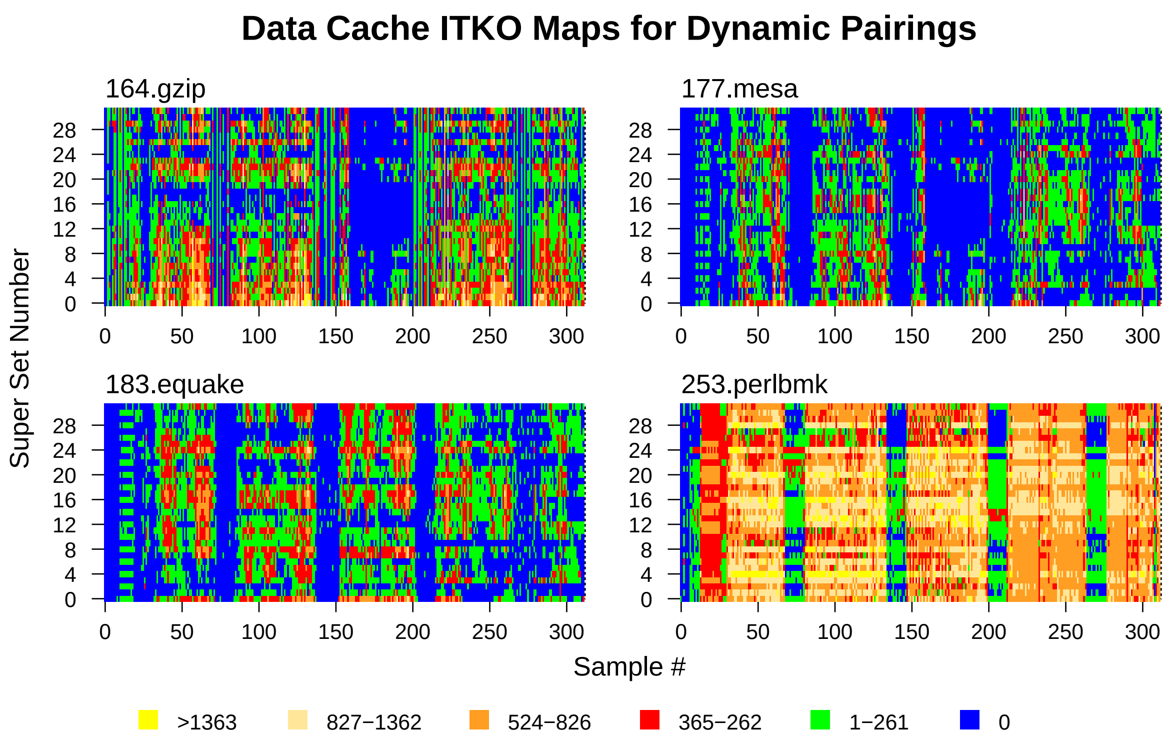


Figure 7: Level 1 data cache Inter-thread interference maps of benchmark pairings using dynamic scheduling.

scheduling periods, the rate of memory instructions, and the necessary precision of the decision point (as discussed in Section 5.1). Finally, a simple incrementing and saturating adder is used to increment the appropriate activity counter when a cache request occurs. The high order bits of the activity counters are then concatenated to create the activity vector for each context. The only potential overhead from this hardware is due to increased fanout on the cache request bus. However, since the hardware executes in parallel to the cache, it can be placed anywhere along the bus where there is extra fan-out capacity.

4. Methodology

4.1 Full System Simulation

The next step in the investigation was to implement the activity vector mechanism in a full system simulator. The simulator used was a version IMPACT LSIM [24] modifi ed to support SMT called Xsim. The simulator was set up as a two-way SMT with a two level cache hierarchy as described in Table 1. The unifi ed level-two cache was divided into thirty-two super-sets. Because of the length of cycle accurate simulation, the scheduling interval was reduced to one million cycles and the simulations were run for one billion cycles in total. The disadvantage of a short scheduling interval is that compulsory misses, brought on by context switches, take on a much larger role in the overall

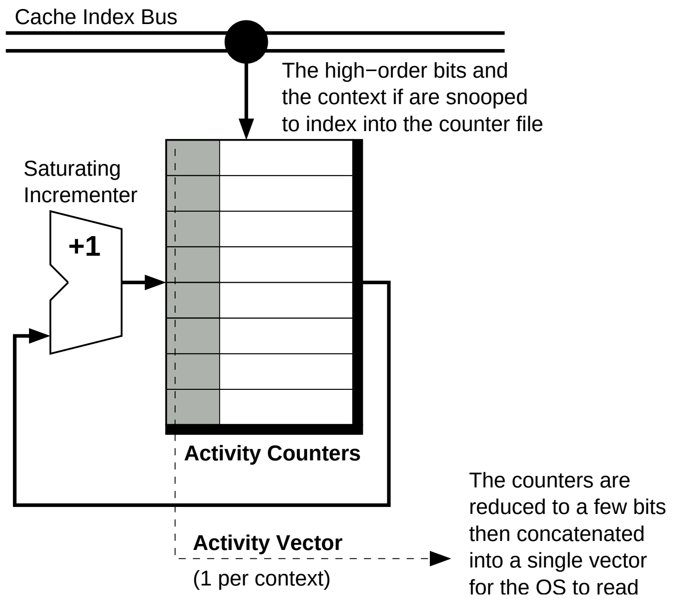

Cache Index Bus
The high−order bits and the context if are snooped to index into the counter file
Saturating Incrementer
+1
Activity Counters
Activity Vector
(1 per context)
The counters are reduced to a few bits then concatenated into a single vector for the OS to read

Figure 8: High-level hardware diagram of fi ne-grained cache activity monitoring hardware.

cache behavior. Scheduling decisions are made on a combination of level two activity vectors and a round robin scheme to ensure fairness. At the end of each sample, the currently resident thread which has been scheduled the most times is scheduled for removal. Each thread is given a score based on the number of common hot regions with the resident thread and the number of periods for which it has already been scheduled. The thread which has the lowest score (this may be the second thread already resident in the processor) is then scheduled for the next interval.

Level	Level Size	Associativity	Block Size
I-Cache	16KB	Direct-mapped	128 Bytes
D-Cache	8KB	2-way	64 Bytes
L2-Cache	512KB	8-way	128 Bytes

Table 1: Xsim Cache Confi guration

Sixteen different combinations of *Spec2000* benchmarks, each consisting of four threads, were tested. The number of available threads was limited because adding more programs increases the number of possible combinations and reduces the reproducibility of the results. Despite the short scheduling interval and limited number of available threads for the scheduler to choose from, the results were quite promising. On average, the number of ITKO was reduced by 10.2% over a round-robin scheduler with one combination showing a 54% improvement. This resulted in an average increase in IPC of 4.4% with a top gain of 12.1%. The results and methodology of these simulations are covered more completely in [25]. These promising results served as motivation to investigate these techniques more completely in a full system.

4.2 Linux Operating System Thread Scheduling

The Linux kernel version 2.6.0 was modified to support activity based job scheduling. The scheduler is run on an Intel Pentium-4 Xeon processor with HyperThreading enabled ([26]). This processor has two contexts, or virtual CPUs, that share the cache hierarchy, data registers, and functional units. The default Linux scheduler runs a symmetric multiprocessing (SMP) scheduling algorithm on each virtual processor to select the next job to run. This algorithm was adjusted to include activity vector information in the job selection criteria. Rather than treat the activity vector information as a score in the scheduling priority algorithm, the default scheduler is first used to select the best job to run. Then, it queries the activity vector scheduler to search for a job that is expected to have the fewest resource conflicts (common hot cache regions, measured by common activity vector bits) with the job running on the other virtual processor. If the activity scheduler cannot identify a better job, then the default job is allowed to run. Since the Pentium4 processor used in these experiments cannot be modified, activity vector support is modeled in software. Because of the large amount of data which needed to be stored and imported into the kernel space, each activity counter is reduced to a single bit.

In order to expose cache activity information to the operating system, each benchmark application is profiled to generate activity vector information. Thus, in order to approximate the actual memory behavior of a given job, each benchmark application is run under the Valgrind memory simulator. Valgrind provides a model of the actual cache hierarchy of the host machine along with a record of runtime program information, such as the instruction count. This simulation stage is used to record all memory requests and misses associated with a given cache super-set. The access and miss counts for each super set are recorded in software activity counters every 50 million instructions. As described in Section 3, a bit is set in the activity vector if the corresponding counter exceeds a threshold value. The resulting activity vectors for each cache level are written to a text file. These vectors are later passed to the operating system kernel memory space, where they are referenced by the modified job scheduling algorithm. Because activity counters were recorded based on the number of completed instructions, finding the activity vector for a program at a given time is simply a matter of knowing the number of completed instructions for a program, which is read from an existing performance counter on the Pentium4.

Algorithm 1 is a pseudo-code representation of the activity based scheduling module and its interface to the kernel scheduling software. The scheduling module is a tool that provides a user level interface to the kernel scheduler, which can be used for both collection of statistics as well as the activation of the scheduling algorithms discussed in this paper. The module was designed so that it can be recompiled and run without rebuilding the Linux kernel. To achieve this goal, the module exports 2 functions to the Linux *schedule()* function. At runtime, the scheduler queries these functions in order to select which job to activate at each schedule invocation. The first function, *th_sample_cache_vector()*, shown in Algorithm 2, is used to sample hardware performance counters and record their associated data for each of the jobs running on the system. The second function, *th_schedule_cache_vector()*, shown in Algorithm 3, is used to select the next job to run based upon analysis of the cache activity vectors.

The *schedule()* function of Algorithm 1 has two important code regions that were chosen as entry points for the activity based scheduling module. The first section is labeled *need_resched*. Here the available tasks either get removed from the run queue if they have completed, or their scheduling priority gets updated. The *th_sample_cache_vector()* function was inserted into this section to mon-

Algorithm 1 Modifi ed linux schedule() function.

```
Require: current_active_task
Ensure: next_task_ready
  if need_resched then
    log_task_runtime()
    th_sample_cache_vector($task$)
    update_context_switch_count()
  end if
  if pick_next_task then
    if runqueues_not_balanced then
      load_balance()
    end if
  end if
  $next\_task$ = select_highest_priority_task()
  $next\_task$ = th_schedule_cache_vector($next\_task$,
      $other\_cpu\_runq$, $task\_priority\_list$)
  recalc_task_prio($next$, $timestamp$)
  if switch_tasks then
    unmodifi ed linux source
    ...
  end if
```

Algorithm 2 Performance monitoring function (th_sample_cache_vector()).

```
Require: current_running_task
Ensure: Global perf counter tables updated
  if $logging\_l2cache\_perf$ then
    sample_perf_counter(L2CACHE, $cpu\_id$)
    log_counter($bmark\_name$)
  end if
  sample_perf_counter($icount$, $cpu\_id$)
  store_icount($task\_id$)
```

itor the state of the cache system performance counter registers associated with the presently active tasks on each of the logical CPUs. The performance counter information was stored internally in the kernel memory space for use in both statistical reporting and as a reference point for the scheduler optimization. The second important region in the function *schedule()* is named *pick_next_task()*. Here, as the name suggests, the kernel selects the next job to activate on the CPU that the scheduler is currently running. The function *th_schedule_cache_vector()* was introduced here to select the best job to run based upon the activity vector information.

Algorithm 2 is very simple. If the level two cache performance is being monitored, the level two cache performance counters are sampled. The second step is to read the instruction count performance counter and update the total number of instructions executed for the thread. This is necessary so that the performance can be aligned with the Valgrind profi le. Algorithm 3 illustrates *th_schedule_cache_vector()*. Because the instruction counter is incremented after the data has been loaded, the activity vector represents the activity in the previous scheduling interval and the only prediction model used is that activity would remain approximately constant between samples. For each of the tasks on the run queue, the default weight is combined with the activity vector intersection (the number of super sets where each thread has high activity) in each level of the cache. The task returned is simply the task which has the lowest weight. The default priority and the activity vector data were weighted such that fairness is preserved and thread starvation avoided.

Algorithm 3 Vector scheduling algorithm (th_schedule_cache_vector()).

```
Require: ready_task, runq_other_cpu,
  active_task_other_cpu, task_priority_array
Ensure: next_active_job
  initialize_job_weights()
  get_vectors($ready\_task$, $other\_task$, $icount$)
  for all Cache_Levels do
    $res\_vec$ = AND($vector\_ready$, $vector\_other$)
    $overlap\_bits$ = count_result_bits($res\_vec$)
    update_weight_function($overlap\_bits$)
  end for
  for all tasks_in_run_queue do
    compare_vectors($task$, $task\_other\_cpu$)
    if $weight < best\_weight$ then
      $best\_task = last\_job\_in\_queue$
    end if
  end for
```

After completing the profiling stage for each of the benchmarks, the user level profiling tool is used to copy the contents of the vector files into the kernel memory. This tool is a device driver that enables a user to control when the scheduler should activate the *th_sample_cache_vector()* and *th_schedule_cache_vector()* functions. When the user interface is invoked, the default Linux scheduler is active. The user is prompted for the file names of the activity vector files, then each is read into kernel memory for later use in the scheduling algorithm. Once the vector files have been read in successfully, the user selects the desired scheduling algorithm, and then starts all of the benchmark applications in the workload set. The total execution time for the workload set is recorded, at which point, the user can command the kernel to return to the default scheduling mode. In addition to enabling activity based scheduling, the user can initiate the performance monitoring routine of Algorithm 2 to record runtime performance statistics.

The overhead of the scheduling algorithm is minimal. In the experimental system the overhead is overstated because the activity vector information must be read from a file and this data must be aligned to the current execution. In a system with the activity vector hardware, neither of these steps would be necessary. The scheduling code would also be more seamlessly integrated as part of the kernel. Additionally, because scheduling decisions are only rarely made (approximately every one hundred milliseconds in Linux), the overhead of scheduling decisions in only a very small percentage of overall execution. In [27], it was shown that most of the overhead of a context switch was due to compulsory cache misses, which will occur regardless of scheduling algorithm. Because the algorithm only chooses one of the threads to run, the overhead of the algorithm scales linearly with the number of tasks on the run queue (each possible thread is compared to the running thread). Unfortunately, the algorithm grows exponentially with the number of contexts which must be filled. This problem is mitigated by the fact that the number of contexts which share a cache will remain low for the foreseeable future. Further, only one or a very few threads should be switched at a given scheduling interval to minimize the number of compulsory cache misses associated with context switches. Since activity vector hardware would be implemented in much the same way as existing performance counters, reading an activity vector would likely have a similar amount of overhead to reading any other performance counter. Our experiments with logging performance counter data at scheduling intervals have shown overheads of less than .1%. This includes the overhead of file I/O which would not be necessary for activity vector based scheduling.

5. Results

5.1 Correlating Cache Activity and Interference

Once a prediction has been made as to how much activity will occur in the next sample for each thread, the scheduler must determine how much interference will occur. The interactions between activity and interference are fairly complex. For example, interference in one level of the cache will lead to increased misses and hence activity in the next lower level of the cache. The most obvious interaction is that increased activity will lead to increased interference as the capacity of a given cache region is exceeded. The total activity across the threads, however, will not differentiate between cases with relatively even usage between the threads and cases where activity is greatly unbalanced. Another possible model is to assume that the interference will be tightly correlated to the amount of activity in the less active thread. The basis of this idea is that the more active thread is in essence saturating the cache region and whatever activity the second thread has will lead to interference. These two models are compared in Figure 9. In these graphs, the level two cache activity is monitored in thirty-two super-sets at one million cycle long intervals across all possible pairings of nine of the *Spec2000* benchmarks (listed in Figure 1). On the left, the total number of misses for both threads is correlated to the number of ITKO in a given super set for a given sample. The correlation of the linear fit, although not ideal, is fairly high. Inspection of the data indicates that the relationship is close to linear, and that total activity is a better indicator at higher activity levels. This is most likely the result of the super-set being saturated in these cases. When all of the space in the region is filled, any additional activity will likely lead to interference. On the right is a similar graph, except in this case the x-axis shows the number of misses in the less active thread. The correlation of the linear fit is significantly higher than that of the total activity and this model seems to perform better at lower activity levels. It is difficult to determine based on this graph exactly how well the model performs at higher activity levels because the data is more sparse in those regions. The fact that the data is concentrated at low activity levels and the minimum is a better indicator at those levels probably has a great deal to do with the higher overall correlation. Since the two models excel for different levels of activity, the ideal model is likely a combination of the two which uses the minimum value for low activity and uses the total value when activity is high.

Another illustration of data can be found in Figure 10. In this graph, the axes each indicate the number of misses in a given super-set and sample. The color is indicative of the median number of ITKO for all samples which have that activity level. Since this is the combined data from all pairings, it is arbitrary which thread is called *A* and which is *B*, so the data is folded over to form the

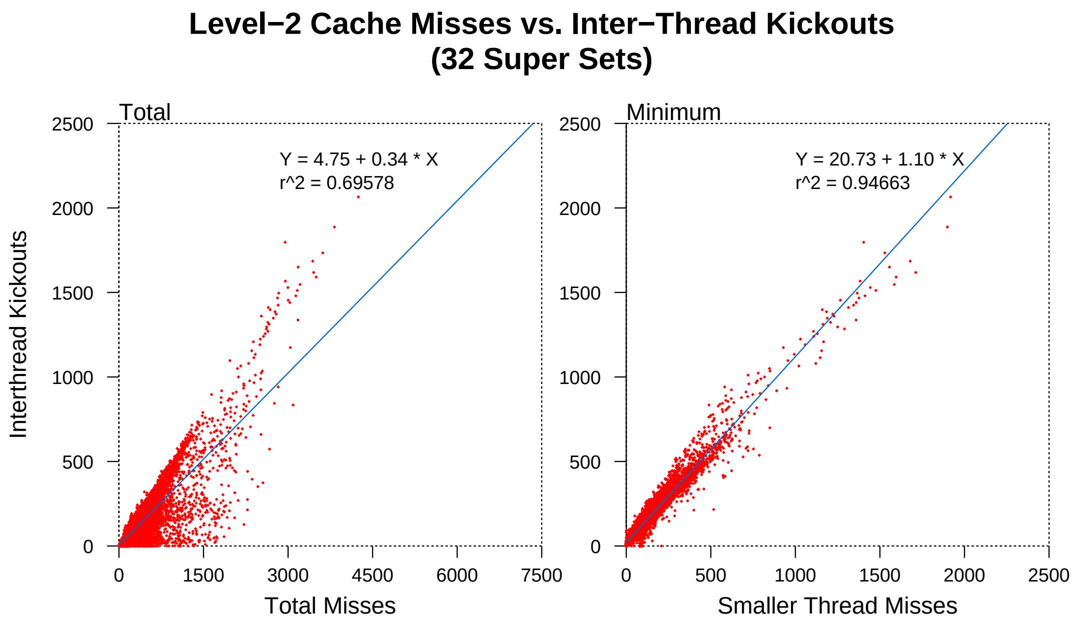
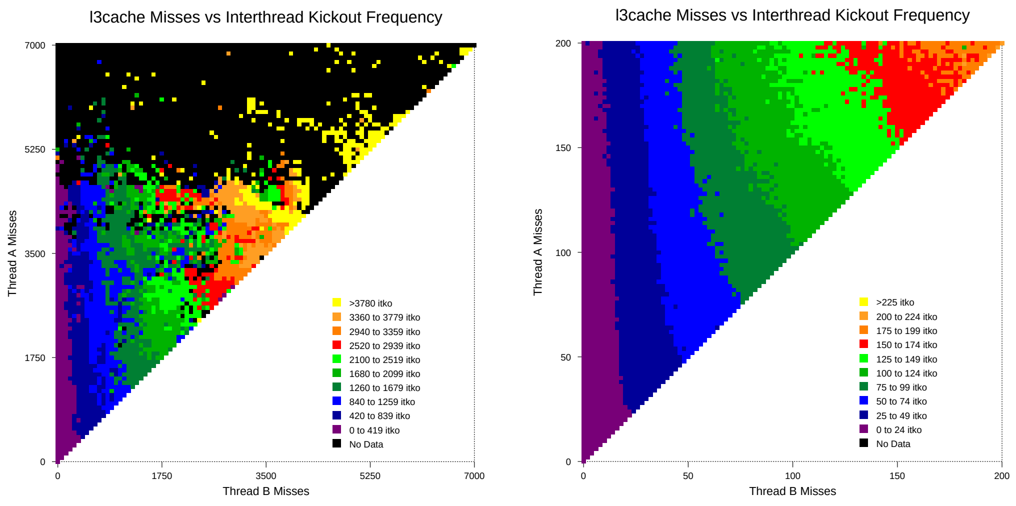


Figure 9: Correlation between number of ITKO and total and minimum number of cache misses.

upper-triangular region shown. On the left is data for the total number of misses across the cache in a given sample and on the right the cache is divided up into thirty-two super-sets. If the minimum model perfectly fi t the data, the graph would consist of evenly-spaced, vertical bars. If the total activity model was ideal, the graph would consist of downward sloping, diagonal bars. Obviously, neither model works perfectly. At low activity levels, the minimum model seems to fi t very well as indicated by the vertical bars toward the left side of the graph. However, at higher levels, the graph more closely resembles the diagonal lines that indicate the sum is the best model. Another important facet of these graphs is the relative data sparsity and lack of clear region boundaries on the graph for the cache total. This demonstrates the large variance in possible activity levels across the cache which make it diffi cult to accurately derive interference from total activity. This is a strong argument for fi ner granularity in cache monitoring.

5.2 Activity Counter Precision

Another important question is how much precision is needed in the activity counters or how many activity levels will be presented to the OS for a given super set. In previous sections, activity was divided into a binary *high* or *low*. This section explores what additional benefi t can be gained by having fi ner granularity in the counters. Since the activity data is inherently noisy, having very high precision is meaningless. Additionally, higher accuracy means that more information needs to be stored and processed by the operating system for each scheduling decision. Conversely, more data may reveal subtle but important variations in behavior that less precise counters may miss.



(a) Entire Cache

(b) 32 Super Sets

Figure 10: Median inter-thread kickouts versus cache misses in the Level 3 cache for 32 super sets and entire cache.

However, Figure 11 illustrates that higher precision is not advantageous beyond one or two bits. The horizontal axis is the number of possible activity levels, and the vertical axis is the average correlation of total misses (quantized to the number levels) to ITKO for that level of the cache. The precision ranges from a single bit up to fi ve bits of precision. Although the data varies from cache to cache, there is no increase in correlation, and hence no extra data nor advantage to be gained from using more than two bits of precision. This and other questions on the details of implementation are more fully explored in [28].

5.3 Memory Aware OS Scheduling

Workload	Benchmark Confi guration	Makeup
WL_1	parser.gap.vortex.bzip2.vpr.mesa.crafty.mcf	Integer
WL_2	mesa.twolf.vortex.gzip.gcc.art.crafty.vpr	
WL_3	gzip.twolf.vpr.bzip2.gcc.gap.mesa.parser	
WL_4	twolf.bzip2.vortex.gap.parser.crafty.equake.mcf	
WL_5	gzip.vpr.gcc.mesa.art.mcf.equake.crafty	Mixed
WL_6	equake.crafty.mcf.parser.art.gap.mesa.vortex	

Table 2: Linux workloads.

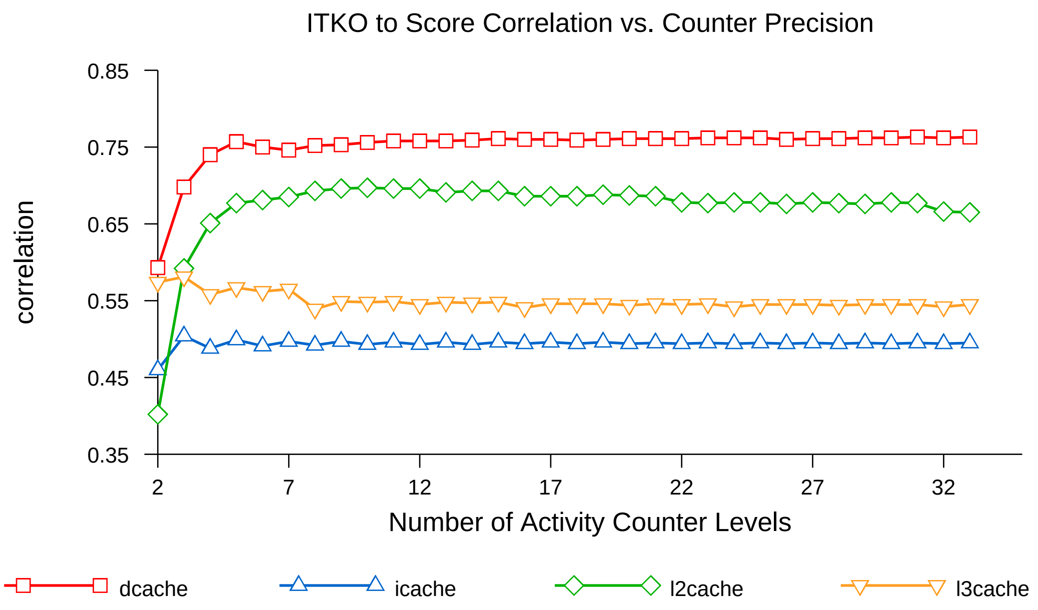


Figure 11: Correlation between number of ITKO and miss activity score for various levels of counter granularity.

The results reported in this section were collected from a set of fi ve batches of eight benchmarks each from the *Spec2000* suite. The benchmarks used in each workload are shown in Table 2. The workloads were chosen to mix memory intensive applications with execution limited ones. Workloads one through four are predominantly integer benchmarks and workloads fi ve and six have an even mix of floating point and integer benchmarks. The benchmarks were all run to completion using the training input set. The use of the training input set was required so that the Valgrind memory profi ling stage could be completed in a reasonable amount of time and the profi le fi les would remain a manageable size. All the jobs were started at the same time and set to run in the background so that the OS could manage their scheduling priorities. In addition to these jobs, the operating system scheduler also manages the system level tasks that run in the background on any Linux system.

The activity based Linux scheduler is confi gured to collect runtime scheduling statistics based upon the hardware performance counters. The cache activity information is used to evaluate how well the job scheduler performs at minimizing potential resource conflicts. Figure 12 shows the percentage of scheduling periods in which the default scheduler chose jobs that were classifi ed as either ‘good’, ‘moderate’, or ‘poor’, relative to the resulting resource contention. These classifi cations were derived by comparing the cache activity of each job in the pool with the activity of the job running on the other virtual CPU. Rather than using this information to change the schedule, it is instead used to assess the effectiveness of the default scheduler. The scheduling decisions of the default Linux algorithm are considered ‘good’ if the activity scheduler would have selected the same job. ‘Bad’ scheduling decisions correspond to cases where the job selected has a high level of interference with the job running on the other CPU. Decisions are rated ‘middle’ if the job that

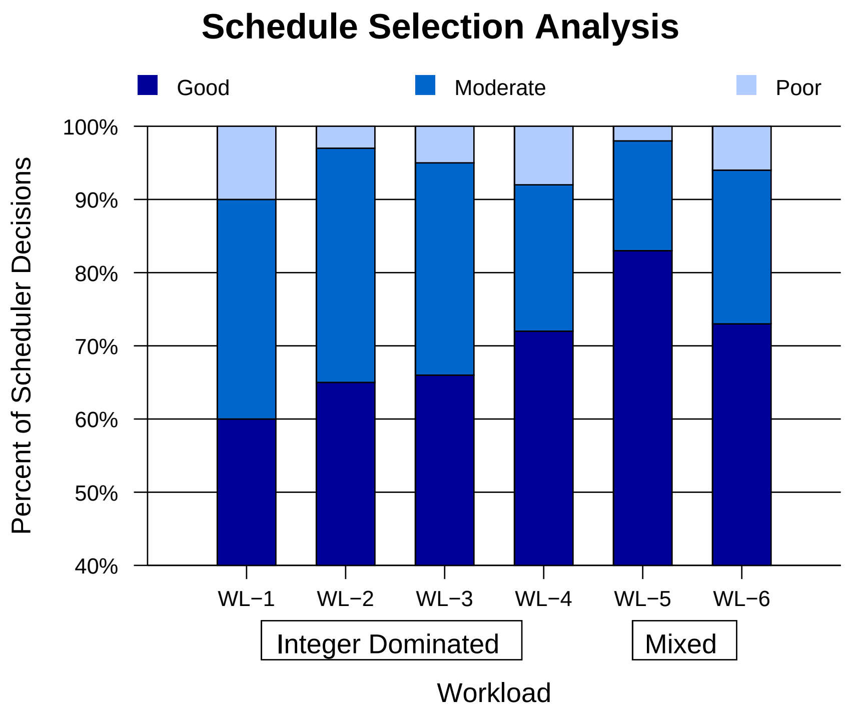


Figure 12: Linux scheduler decisions.

was selected differed from the activity vector scheduler, but the level of interference was moderate. From Figure 12, one can see that for each of the 6 workloads, the default scheduler makes a poor scheduling decision for 30%-40% of the scheduler invocations. Therefore, there is signifi cant room for improvement for scheduling tuned to reduce resource contention. In addition, as the number of jobs in the pool increases, the opportunity for the activity based scheduler to fi nd a good match for co-scheduling should also increase.

After analyzing the default scheduler, the activity based scheduler was enabled and the resulting performance for the set of workloads was measured and reported in Figure 13. For each workload, the L2 cache misses associated with the workload were recorded along with the IPC. Four out of the six workloads show an improvement in L2 cache misses of more than **4%**, with a similar improvement in IPC. The net performance gain is dependent upon the job mix, which is expected since some of the workloads may contain a number of jobs with similar cache activities. The results are encouraging since this technique was implemented on real hardware with a commonly used operating system.

These results are pessimistic for several reasons. As discussed in Section 4.2, the overhead of the scheduler, which is included in the total runtime presented above, is overstated. The prototype implementation must read in the pre-profi led memory data and align it with the current point in the execution, which would not have to be done if the activity vector hardware were available, saving

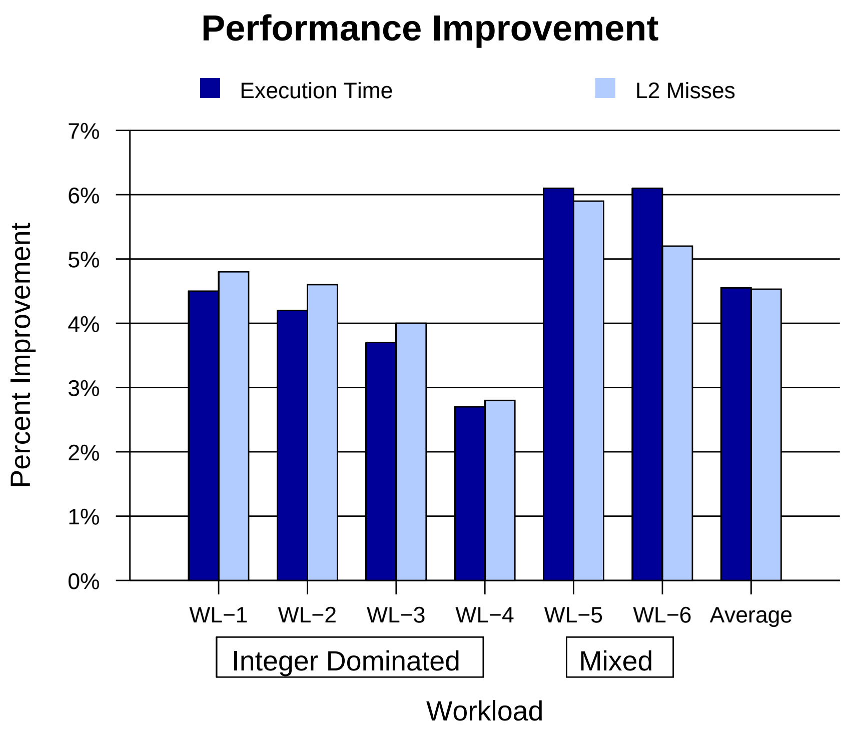


Figure 13: Performance improvement using activity vector scheduling.

the overhead of both the file I/O and alignment. A full implementation could also further optimize and integrate the scheduler into the kernel. Additionally, the use of the training inputs, necessitated by the use of pre-profiling, decreases the run times, and hence the scheduling opportunities available to the OS. Since the activity vector information facilitates better scheduling decisions, the benefit is directly related to the number of decisions made. Finally, the use of Spec2000 benchmarks is probably more taxing than a real world system would be. Most commonly used benchmarks are large programs designed to tax various components of the system. In a real system there are typically only a few large programs running with a variety of smaller system tasks running in the background. One additional point is that the system tasks which ran during the tests are not profiled, and therefore were scheduled naively. Since the activity vector hardware is independent of the workload, these could also be more intelligently scheduled with activity information.

6. Related Work

6.1 Multithreading

Multithreading processors are becoming the industry standard for high performance computer systems. Numerous thread models exist which emerged as solutions to specific performance issues.

The initial implementations were designed to improve the performance of shared memory multi processor computer systems. Researchers observed that individual nodes were spending significant portions of their execution time idle waiting to service off-chip memory accesses. The APRIL processor from MIT was a coarse grain multithreading (CGMT) implementation of a single processor that could execute instructions from a different software context while the main context was servicing a memory request [17]. Gupta and Weber [18] were attracted to CGMT for similar reasons; they had identified the increasing memory latency due to remote memory accesses in a multiprocessor system. They observed that the performance improvement due to multithreading was application dependent, and was also limited by the interference introduced by the addition of instructions from other hardware contexts. The first commercial implementation of a coarse grain multithreading processor is the IBM Power PC RS64 IV [29].

The introduction of fine grained multithreaded processors preceded the widespread use of superscalar architectures. In [30] the fine grained thread model was motivated from the need to increase the utilization of processors with deep pipelines. Rather than perform control speculation and out of order execution to maintain a steady supply of independent instructions, Farrens and Pleszkun proposed alternating each issue cycle with instructions from a different process. Interleaving the instructions from independent contexts provided time for the processor to resolve register dependencies and other hazards associated with each program. Reducing these dependencies prevented pipeline stalls from occurring and led to an increase in processor utilization.

Simultaneous multithreading later emerged as a low cost means to improve the utilization of superscalar processors. Wide issue superscalar processors proved effective for exploiting instruction level parallelism, however control and data dependencies limited the number of independent instructions available in a given instruction sequence. By enabling the issue of instructions from multiple contexts each cycle, periods of low instruction level parallelism (ILP) could be compensated for by thread level parallelism (TLP) [3, 5, 1]. Simultaneous multithreading has since emerged as a popular platform for high performance processors, and has proved very useful for computing environments where several jobs are run in parallel.

Speculative multithreading recently emerged as a means to leverage the additional processing resources introduced by SMT to advance the performance of single jobs. Numerous researchers have investigated this area, and have proposed a variety of techniques for exposing additional parallelism in a serial job. Chappell, *et al.* introduced simultaneous subordinate multithreading [31] as the starting point for speculative threading. Others have proposed entirely new architectures such as the multiscalar architecture introduced by Sohi in [32]. From this field, two important areas have emerged. The first is to use speculative threading for extracting additional parallelism from a single application. Marcuello and Gonzalez [33, 34, 35] pioneered much of the work in this area. Their approach achieves performance improvements by increasing the amount of ILP available to the processor hardware. The other field uses speculative threads as a form of data and instruction prefetching. Speculative precomputation, for example [36, 35], uses the compiler to identify long latency load instructions. Then, speculative threads are used to execute the instructions that compute the load address and the corresponding load. The speculative threads effectively initiate prefetches for the main thread. Other techniques such as the Slipstream architecture employ a similar strategy of using speculative threads to assist the main thread [37]. Wang, *et al.* demonstrated a virtual multithreading technique where lowest level cache misses caused specially chosen code from the same thread to execute, effectively hiding memory latency [38].

6.2 SMT Performance Studies

There have been numerous studies focused on improving the throughput performance of simultaneous multithreading processors. The techniques for solving this class of problems are covered by a range of disciplines, including operating system thread scheduling, dedicated architecture techniques, and cooperative compilation techniques [39]. Although the SMT improves the throughput of multi-job workloads, performance bottlenecks related to the shared hardware resources on these processors have placed limitations on their effi ciency. In particular, the shared cache hierarchy [40], and the instruction fetch and issue logic [41, 42] have emerged as components that require special attention in an SMT environment.

In response to the importance of resource contention between threads associated with the cache hierarchy, there have been a number of operating system level solutions to solving this problem. Lo, *et al.* demonstrated in [43] that database applications which suffered from inter-thread conflict misses could be improved signifi cantly by using the operating system to change the way virtual memory pages were mapped to their physical counterparts. This approach made it less likely that data belonging to different thread contexts would map to the same cache sets, thus reducing the amount of inter-thread conflict misses. While the results from this study are encouraging, the approach focuses on database applications which have fundamentally different data footprints than workloads studied in this paper (Table 2). Symbiotic job scheduling [14] was introduced by Snavely and Tullsen as a method to improve performance by making the operating system co-schedule threads that were likely to 'behave' well with one another. While this technique exposed the potential headroom for using the operating system to advance SMT performance, it did not reflect closely enough the runtime environment of a real operating system job scheduler. Chandra, *et al.* presented a mathematical model for predicting the interference between threads in a CMP based on reuse distance in [44].

Parekh, Eggers, and Levy proposed exposing microarchitecture level performance information to the operating system job scheduler in [15]. This work showed IPC improvements on the order of 10%, however, the experiments were conducted on a simulated operating system and processor architecture. Suh *et al.* used fi ne grained cache monitoring to partition the cache based upon the predicted demand from each thread in [9]. The resource demands of each thread are determined by analyzing the frequency and time of last use for each cache line. The cache is then partitioned among the threads, each one allocated a number of sets proportional to its expected resource demands.

Studies of the impact of compiler optimizations on SMT architectures in [45] show that performance improvements can be achieved provided the optimizations are modifi ed to account for the SMT microarchitecture. Specifi cally, compiler optimizations that are used to expose ILP on a uniprocessor often lead to increased code size, such as with loop unrolling. In an SMT processor, thread level parallelism between processes may be suffi cient to maintain high utilization, making the ILP optimizations unnecessary. Kuhmar and Tullsen investigated compile time methods for reducing the amount of interference between threads in the instruction cache in [46]. This work focused primarily on the i-cache and is specifi cally aimed at static, rather than dynamic optimization techniques.

6.3 Working Set Analysis

Research on the working sets of programs stretches back over thirty-fi ve years ([47, 48]), however most studies work only on the level of operating systems and paging. Page coloring ([49]) is a commonly used technique to map virtual pages to different regions of the cache and distribute the demand across the cache from a given thread. Takir and Hollingsworth ([50]) used the bus monitoring hardware on a multiprocessor server to migrate pages to the local memory of the processor most closely associated with the data. Sherwood, Calder, and Emer ([51]) proposed a methodology similar to the one presented in this paper to monitor individual regions of the cache. This information was used in a single-threaded environment to remap pages away from hot regions and more evenly distribute cache demand across the cache. Applying these techniques to a multithreaded environment is an area of ongoing research.

7. Conclusions

7.1 Future Work

Exposing cache activity vector information to the operating system scheduler has proved to be effective at improving the performance of SMT workloads. However, as this work has demonstrated, the problem of capturing inter-thread interference information at runtime remains a challenge. As such, future work in this area will be directed at improving the techniques for detecting this interference. Additionally, while measuring inter-thread kick outs is fairly simple to do with a simulator, it is quite challenging to implement in real hardware. Thus we are exploring the correlations between other event counters and ITKO in order to derive an accurate approximation of an ITKO hardware counter. These techniques may be applicable to other multithreading paradigms and CMP. We anticipate that with improved inter-thread interference detection schemes, the operating system scheduler should be able to further improve the performance of SMT workloads. Further, fi ne-grained cache monitoring can be used for page remapping ([51]) to mitigate inter and intra-thread interference. With the activity vector already available to the OS for scheduling, we are investigating page remapping based on this information to complement the scheduling decisions.

7.2 Summary

SMT offers a promising method to circumvent the limitations on ILP imposed by such factors as long latency operations and memory accesses and their related data and control dependencies by exploiting TLP. However, there are serious limitations imposed by contention for limited resources between threads. Perhaps the most important of these resources is memory hierarchy capacity. Because inter-thread kick outs produce extra cache misses, the performance of individual threads is degraded, counteracting the overall effectiveness of SMT. By carefully monitoring and predicting the access patterns of each thread in an SMT system and exposing this information to the operating system scheduler, intelligent scheduling decisions can be made that minimize the amount contention for cache space. This minimization is accomplished by choosing threads that access different regions of the cache, and thus can operate in parallel, even though each thread may have high overall demand on the cache. A modifi ed thread scheduler, using simulated fi ne-grained information is able to signifi cantly reduce cache contention between threads and improve performance on commercially available hardware and thus recapture much of the parallelism lost to interference between threads.

Acknowledgements

The authors would like to thank Tipp Moseley and Josh Stone for their assistance with Valgrind and the Linux Scheduler. Computer time was provided by NSF ARI Grant #CDA-9601817, NSF MRI Grant #CNS-0420873, NASA AIST grant #NAG2-1646, DOE SciDAC grant #DE-FG02-04ER63870, NSF sponsorship of the National Center for Atmospheric Research, and a grant from the IBM Shared University Research (SUR) program. The authors would also like to thank the anonymous reviewers for their helpful insights which have greatly improved this work.

References

[1] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, vol. 17, pp. 12–18, September/October 1997.

[2] V. Krishnan and J. Torrellas, "A chip-multiprocessor architecture with speculative multithreading," *IEEE Transactions on Computers*, vol. 48, no. 9, pp. 866–880, 1999.

[3] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *22nd Annual International Symposium on Computer Architecture*, June 1995.

[4] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy, "Supporting fi ne-grained synchronization on a simultaneous multithreading processor," in *International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 54–58, 2000.

[5] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen, "ILP versus TLP on SMT," *Supercomputing*, November 1999.

[6] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 36–44, June 1985.

[7] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, "Compiler technology for future microprocessors," *Proceedings of the IEEE*, vol. 83, pp. 1625–1995, December 1995.

[8] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proceedings of the 23rd annual International Symposium on Computer Architecture*, pp. 191–202, 1996.

[9] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proceedings of the Eigth International Symposium on High Perfromance Computer Architecture (HPCA)*, pp. 117–128, 2002.

[10] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2004 (PACT '04)*, pp. 111–121, IEEE Computer Society, 2004.

[11] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernendez, A. Ramirez, and M. Valero, "Predictable performance in smt processors," in *CF'04: Proceedings of the First Conference on Computing Frontiers*, pp. 433–443, ACM Press, 2004.

[12] I. Corporation, "Special issue on intel hyperthreading in pentium-4 processors," *Intel Technology Journal*, vol. 1, January 2002.

[13] S. Hily and A. Seznec, "Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading," Tech. Rep. PI-1086, IRISA, 1997.

[14] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *Architectural Support for Programming Languages and Operating Systems*, pp. 234–244, 2000.

[15] S. Parekh, S. Eggers, and H. Levy, "Thread-sensitive scheduling for smt processors," tech. rep., Department of Computer Science & Engineering University of Washington, Seattle, Washington, 2000.

[16] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfi eld, and B. Smith, "The Tera computer system," in *Proceedings of the1990 International Conference on Supercomputing*, pp. 1–6, 1990.

[17] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz, "APRIL: A processor architecture for multiprocessing," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, (Seattle, WA), pp. 104–114, 1990.

[18] W. Weber and A. Gupta, "Exporing the benefi ts of multiple hardware contexts in a multiprocessor architecture: Preliminary results," in *Proceedings of the 16th Interational Symposium on Computer Architecture (ISCA)*, pp. 273–280, June 1989.

[19] M. V. T. Sherwood and B. Clader, "A co-phase matrix to guide simultaneous multithreading simulation," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2004.

[20] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to fi nd periodic behavior and simulation points in applications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, 2001.

[21] B. Porat, *A Course in Digital Signal Processing*. New York: John Wiley and Sons, Inc., 1997.

[22] J. Kihm, A. Janiszewski, and D. Connors, "Predictable fi ne-grained cache behavior for enhanced simultaneous multithreading (SMT) scheduling," in *Proceedings of International Conference on Computing, Communications and Control Technologies*, 2004.

[23] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz, "Multi-processor performance on the tera mta," in *SC '98: Proceedings of the Proceedings of the IEEE/ACM SC98 Conference*, p. 4, IEEE Computer Society, 1998.

[24] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[25] A. Settle, J. Kihm, A. Janiszewski, and D. A. Connors, "Architectural support for enhanced SMT job scheduling," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, IEEE Computer Society, 2004.

[26] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Santa Clara, CA, 2004.

[27] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," in *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pp. 75–84, ACM Press, 1991.

[28] J. Kihm and D. Connors, "Implementation of fi ne-grained cache monitoring for improved smt scheduling," in *Proceedings of The 22nd International Conference on Computer Design*, 2004.

[29] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel, "A multithreaded powerpc processor for commercial servers," *IBM Journal of Research and Development*, vol. 44, pp. 885–898, November 2000.

[30] M. K. Farrens and A. R. Pleszkun, "Strategies for achieving improved processor throughput," in *Proceedings of the 18th annual international symposium on Computer architecture*, pp. 362–369, ACM Press, 1991.

[31] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 186–195, May 1999.

[32] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *25 Years ISCA: Retrospectives and Reprints*, pp. 521–532, 1998.

[33] P. Marcuello and A. González, "Exploiting speculative thread-level parallelism on a smt processor.," in *HPCN Europe*, pp. 754–763, 1999.

[34] P. Marcuello and A. Gonzalez, "A quantitative assesment of thread-level speculation techniques," in *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pp. 595–604, May 2000.

[35] T. Aamodt, P. Marcuello, P. Chow, P. Hammarlund, and H. Wang, "Prescient instruction prefetch," in *Proc. of the 6th Workshop on Multithreaded Execution, Architecture and Compilation*, pp. 2–10, November 2002.

[36] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y. fong Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[37] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[38] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen, "Helper threads via virtual multithreading on an experimental itanium 2 processor-based platform," in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pp. 144–155, ACM Press, 2004.

[39] R. Kumar and D. Tullsen, “Compiling for instruction cache performance on a multithreaded architecture,” in *Proceedings of the 35th International Symposium on Microarchitecture*, November 2002.

[40] S. Hily and A. Seznec, “Standard memory hierarchy does not fi t simultaneous multithreading,” in *in Proc. of the Workshop on Multithreaded Execution Architecture and Compilation (with HPCA-4)*, 1998.

[41] A. El-Moursy and D. Albonesi, “Front-end policies for improved issue effi ciency in smt processors.,” in *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA’03)*, p. 31, IEEE Computer Society, 2003.

[42] K. Luo, M. Franklin, S. S. Mukherjee, and A. Seznec, “Boosting SMT performance by speculation control,” in *Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 2–9, 2001.

[43] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh, “An analysis of database workload performance on simultaneous multithreaded processors,” in *Proceedings of the 25th International Symposium on Computer Achitecture (ISCA)*, pp. 39–50, 1998.

[44] D. Chandra, F. Guo, S. Kim, and Y. Solihin, “Predicting inter-thread cache contention on a chip multi-processor architecture,” in *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 340–351, Feb 2005.

[45] J. L. Lo, S. J. Eggers, H. M. Levy, S. S. Parekh, and D. M. Tullsen, “Tuning compiler optimizations for simultaneous multithreading,” in *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 114–124, December 1997.

[46] R. Kumar and D. M. Tullsen, “Compiling for instruction cache performance on a multithreaded architecture,” in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 419–429, IEEE Computer Society Press, 2002.

[47] P. J. Denning, “The working set model for program behavior,” *Communications of the ACM*, vol. 11, pp. 323–333, May 1968.

[48] P. J. Denning, “Working sets past and present,” *IEEE Transactions on Software Engineering*, vol. SE.6, pp. 64–84, January 1980.

[49] W. L. Lynch, *The Interaction of Virtual Memory and Cache Memory*. PhD thesis, Stanford University, Palo Alto, CA, 1993.

[50] M. M. Tikir and J. K. Hollingsworth, “Using hardware counters to automatically improve memory performance,” in *Proceedings of 2004 High Performance Computing, Networking, and Storage Conference (SC)*, p. 46, 2004.

[51] T. Sherwood, B. Calder, and J. Emer, “Reducing cache misses using hardware and software page placement,” in *ICS ’99: Proceedings of the 13th International Conference on Supercomputing*, pp. 155–164, ACM Press, 1999.