Scalable Compiler Optimizations for Improving the Memory System Performance in Multi- and Many-core Processors

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Sanyam Mehta

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Pen-Chung Yew

September, 2014

Acknowledgements

I would first of all acknowledge my Teachers for molding the direction of my life's path, especially when I had entered my Bachelor's and needed such clear guidance the most. Not only was I introduced to research by them, but was also, more importantly, taught the utility of the education that I was getting. I have ever since been clear about things to do next that will benefit me and others, and thus contribute positively in a little way to the society.

As a PhD student, I feel that I was blessed to have Professor Yew as my advisor. Right from the beginning, he encouraged and helped me to choose the right problems to work on, and was always there to provide me with the right guidance at the right moments based on his vast experience of our research area. In addition, I would also like to extend my heartfelt thanks to Professor Zhai, for helping me refine my work and make it appealing to the target audience. She has especially been very helpful in teaching me to make presentations for conferences. Also, I would thank, Professor Kumar for always being very warm and accomodating throughout my stay at the University of Minnesota and while serving as a member on my committee, and Professor David Lilja, Professor Stephen McCamant, Professor Daniel Boley and Dr. Aamer Jaleel for honoring me with their presence during my thesis defense and asking me insightful questions.

I would also thank my very dear friends, Ankit, Ashish, Ayush, Rajat, Saket, and Shashank for helping me in various ways to stay focused towards my goal in life. It is because of being happy in their presence that I could work effectively at college.

Finally, I would thank my parents, brother, and also other family members. They first of all cared for me and taught me all that I knew until I grew up and could think about things, and then allowing me to be in a different country for my PhD. They have been very understanding and accomodating to allow me to do something that I thought would be helpful for all, and have always encouraged me and blessed me to be successful.

Dedication

To my Teacher and Parents

Abstract

The last decade has seen the transition from unicore processors to their multi-core (and now many-core) counterparts. Today, multi-cores are ubiquitous - they form the core fabric of our laptop and desktop PCs, supercomputers, datacenters, and also mobile devices. This transition has brought about renewed focus on compiler developers to extract performance from these parallel processors. In addition to extracting parallelism, another important responsibility of a parallelizing (or optimizing) compiler is to improve the memory system performance of the source program. This is particularly important because all cores on the chip simultaneously demand for data from the slow memory, and thus computation ends up waiting for the arriving data. In other words, the multi-cores have accentuated the *memory-wall*. These simultaneous requests for data from off-chip memory also leads to contention for bandwidth in off-chip network (called *bandwidth wall*), leading to further increase in the effective memory latency as seen by the executing program.

While the above responsibilities of a parallelizing compiler are better understood, we identify three key challenges facing the compiler developers on current processors. These include, (1) the diverse set of microarchitectures existent at any time, and more importantly, the changes in micrarchitecture between generations. We identify that the existing compilers have particularly been unable to adapt to some of the important changes to microarchitecture in the last decade, resulting in suboptimal optimizations. (2) Poor show of compilers in real applications that contain large scope of statements amenable for optimization. This weakness stems from the lack of a good cost model for deciding statements to fuse, and sheer inability to fuse due to ineffective dependence analysis. (3) Unscalability of compilers - this is a traditional limitation of compilers where the compilers choose to optimize small scopes to contain the compile time and memory requirement, and thus loose optimization opportunities.

In this thesis, we make the following contributions to address the above challenges.

1. We revisit three compiler optimizations (loop tiling and loop fusion for enhancing temporal locality and data prefetching for hiding memory latency) for improving memory (and parallel) performance in light of the various recent advances in microarchitecture, including deeper memory hierarchy, the multithreading technology, the (short-vector)

SIMDization technology, and hardware prefetching, and propose generic algorithms implementable in production compilers for a range of processors.

2. We propose wise heuristics in a cost model to choose good statements to fuse, and also improve dependence analysis to not loose critical fusion opportunity in application programs when it exists.

3. The final contribution of this thesis is a solution to the unscalability problem. Based on program semantics, we devise a way to represent the entire program with much fewer representative statements and dependences, leading to significantly improved compile time and memory requirement for compilation. Thus, real applications can now be optimized not only efficiently, but also at a very low overhead.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Until a decade ago, Moore's law translated into deeper pipelines (and corresponding increase in processor frequency), sophisticated cores with newer technologies for branch prediction and Instruction-level Parallelism (ILP), and larger on-chip caches. All this meant increasing performance gains without any involvement of the programmer or the optimizing compiler. However, this trend has changed for various reasons. It is now no longer possible to increase the clock frequency due to power dissipation issues (1). The era of ILP has also witnessed an end since very little gains from employing transistors for extracting ILP are estimated, and the community has exhausted options in this regard (2). The combined effect has been a shift to *multiple cores* on a chip, where each core runs at a lower frequency to save power. Since the number of transistors can now be increased in accord with Moore's law without excessive power consumption, continued increase in performance can be sustained. This performance increase is, however, contingent on ability of the source program to tap into the parallelism exposed by the hardware.

This marked shift in the processor design has brought about a renewed focus on compiler design. Earlier, the onus of improving performance was borne mostly by the hardware without a significant contribution by the compiler or the programmer. This was natural since performance improvement could be achieved by optimizing execution of instructions within a small window through techniques for extracting ILP. These techniques included dynamic out-of-order execution, superscalar processing, speculative execution and non-blocking caches. However, with the introduction of multi-core processors, extracting parallelism from within a few instructions is not sufficient, and parallelism at a much more coarser level is necessitated. This coarse-grained parallelism is not easy for the hardware to extract since it has a view of a narrow window of instructions. Therefore, the onus of improving performance on multi-core processors has to be now borne by either the compiler or the programmer.

Clearly, extracting coarse-grained parallelism where the compiler directs each core on the host hardware to execute specific portions of the source program, is certainly one important responsibility of the compiler. However, the shift to multicore has also had other important implications which cannot be ignored. In particular, when multiple cores on the chip compute in parallel, they also consume data in parallel. This implies simultaneous requests for data from the off-chip memory. Now, it is well known that the improvement in memory speed has not kept up pace with the processor speed (leading to the memory wall), and multiple cores on chip only help to accentuate this gap between the processor and memory. Another related effect that creeps in and is equally harmful to performance is the bandwidth wall, where effective memory latencies increase due to bandwidth contention resulting from memory-intensive applications and poor memory performance. Thus, alleviating effects of memory wall and bandwidth wall for multi-cores is the second important responsibility of the parallelizing compilers. Again, like parallelization, the hardware cannot excavate opportunties existent in the source programs for any potential improvements in memory performance. It is certainly unrealistic to pass this responsibility to the programmer, because the host platforms tend to be so different and evolving, and NOT all programmers like to be 'close to the silicon' or feel comfortable at reasoning an optimization for long sequences of loop nests at their disposal. In such a scenario, traditional memory optimizations such as loop tiling, loop fusion, and data prefetching performed by a parallelizing compiler assume renewed significance.

1.1 Key challenges for compiler developers in the present day

While the two important responsibilities of a parallelizing compiler are well understood, it is also important to understand the challenges facing the compiler developers when meeting these resposibilities. Our experience with parallelizing compilers and present-day architectures reveals that there are three key challenges in this regard:

1.1.1 The diverse world of computer architecture that also keeps on evolving

The first key challenge in developing a compiler stems from its intimate relationship with the host hardware. For any compiler to yield the best performance when applying a particular optimization, it must have a thorough understanding of the host hardware. For example, the choice of a good tile size depends on the size of the cache, its set-associativity, number of levels of cache, prefetching behavior of the hardware, etc. Similarly, the degree of aggressiveness at which loops should be fused, depends on cache characteristics, vectorization potential of the host hardware, etc. As a result, it becomes difficult for a compiler to perform optimally on the diverse set of microprocessors that exist today. While the characteristics of the contemporary microprocessors from different vendors such as AMD and Intel are largely the same, we identify that an even more important problem is that traditional compiler optimizations such as loop tiling, loop fusion and prefetching as they exist in present-day processors have failed to evolve with the major changes to computer microarchitecture in the last decade (over multiple microarchitecture generations). The changes include the following:

1. **Deep memory hierarchies** - Having recognized the importance of alleviating off-chip memory accesses, the existing multi-cores have seen a shift from single-level, to two-level, to now a three-level cache hierarchy. However, important compiler optimizations such as loop tiling as they exist in current production compilers do not account for this change. Most of the work on loop tiling assumes that processors need to achieve data reuse in a single level of cache, and there is some work that assumes a two-level cache hierarchy, but the optimization is still not tuned for optimal reuse. Similarly, existence of multiple levels of cache and the opportunity to prefetch data selectively to those levels requires a carefully executed strategy for optimal performance in current processors, perhaps with coordination between the compiler and hardware.

2. **The multithreading technology** - In current processors, multithreading is available through either Chip Muti-Processing (CMP) or Simultaneous Multi-Threading (SMT). Both these technologies have a bearing on the memory optimizations performed by the compiler. For example, if 2 threads are running in CMP, then the two threads simultaneously bring the data to the shared last level cache. Similarly, if 2 threads are running in SMT, then they bring in data simultaneously to even the private L1 or L2 cache on each core, and thus reduce each other's share of the cache. Threads in SMT may have an even more involved relationship. For example, in Intel's latest many-core processor, Xeon Phi, a thread can only issue instructions once every 2 cycles, and another thread is granted opportunity for the next cycle. In such cases, the compiler must account for such behavior for optimal performance.

3. **The vectorization technology** - Current processors employ efficient vectorization or

SIMD (Single-Instruction Multiple Data) units to perform parallel computation (called vectorization) on short vectors. The vectorization technology is used to extract the second level of parallelization available in source program which is at a much finer-granularity (generally the innermost loop in a loop-nest) than the coarse-grain or outer-loop parallelism. Current production compilers, particularly the Intel compiler, are adept at finding vectorizable loops because vectorization purchases considerable performance improvement. We recognize that although production compilers are not so good in finding coarse-grain parallelism, they can find fine-grain parallelism because the analysis merely involves a single loop and its body, instead of an entire loop-nest. However, they still fail to study its interaction with other memory optimizations such as loop tiling and loop fusion. For example, if the compiler takes the approach of aggressive loop fusion, then that might hurt vectorization because of introduction of loop-carried dependences in the innermost loop. Similarly, certain tile sizes benefit significantly more from vectorization than others, and thus vectorization has an important say in tile size selection as well. In some cases, tiling may even degrade performance of the source program because of its detrimental impact on vectorization.

4. **The prefetching technology** - Data prefetching is the single most important and generally applicable technology that fetches performance on existing multi- and many-core processors. This is because the gap between the processor and the memory speed is the largest in the present day, and it is therefore required to pre-fetch the data from the slow memory for timely execution. The way this technology interacts with the compiler is that the compiler is also armed with prefetch instructions, and thus software and hardware prefetching contend with each other. To add to this, there are hardware prefetchers for different levels of cache, and so are there software prefetch requests for different levels, and to decide how they should coordinate to achieve the best performance is a live challenge. Certainly, the compiler cannot perform the prefetching optimization oblivious of the hardware prefetcher's abilities and inabilities. It should instead know those and complement the weaknesses of the hardware prefetcher. Also, since prefetching is so crucial to performance, other optimizations such as tiling and fusion must be performed by the compiler so as to not hurt prefetching. For example, tiling induces block-wise execution of the program arrays instead of sequential execution. This hurts prefetching, and thus requires a careful balance. Similarly, aggressive fusion may result in merging many arrays

into the same nest. This increases the number of prefetch streams required to monitor them, and the processor may fall short of them which can potentially hurt performance.

Since such an intimate interaction exists between compiler memory optimizations and the above-mentioned important advances in computer microarchitecture, it is important for the compiler to take into account the host hardware features for best performance. We suggest that the compiler should ideally extract this needed information from the processor's host OS or during its installation, and then use it every time it goes ahead to perform its optimizations. In any case, these optimizations need to be implemented in a compiler in a way that it provides the needed flexibility to adapt to different hosts.

1.1.2 Real applications and poor show

The greatest times of need for a programmer are when the source code is large and requires memory optimizations to improve performance. Such is the case in many real scientific applications, and the compiler's help is inevitably sought. However, our experiments with the state-of-the-art production compilers on such real applications from popular benchmark suites, show that it is in such cases, that the compiler is particularly ineffective. Although there exists immense opportunity to improve temporal locality of data accessed in such applications, the compiler is actually able to do very little to help the programmer in need. We identify that this unfriendliness of the compiler in such circumstances arises from 2 key reasons:

1. The hot subroutines in various scientific applications contain sequences of loop nests. These loop nests tend to access common data and thus contain substantial opportunity for data reuse through fusion. However, the production compilers either simply choose to ignore the possibility of fusion, or choose to be very restrictive in fusing nests. The first important reason is that they only consider very small scopes for fusion, such as just consecutive nests in 'pair-wise fusion'. Thus, if two nests that are not adjacent are fusable, the compiler will simply ignore the possibility. This is a consequence of the compiler's view of reducing analysis time by deciding on small scopes and thus fewer dependences to analyze. Also, the criteria to be chosen for deciding the best fusion structure is unclear especially in the wake of recent changes to microarchitecture as discussed above. For example, if the compiler leads to an imperfect fusion of nests (which results when the fused nests have different loop-depth) with the help of the insertion of conditional clauses in

the loop body, then it could hurt vectorization, and be unprofitable. Thus, a clear criteria or more precisely, a cost model, is necessitated to decide on good fusion structures.

2. The second important problem is that even if the criteria for choosing nests to fuse is decided, the fusion is hindered by the occurrence of certain artificial dependences between the candidate nests due to the frequent use of temporary variables in such large scientific applications. The use of such temporary variables, both scalars and arrays, arises from the fact that these applications compute partial results to be immediately used in the program. Those partial results are stored in temporary variables to avoid re-computation. Existing production compilers, because of their oversight of the importance and opportunity of fusion in such applications, tend to ignore the presence of such fusion-preventing dependences. These dependences are artificial because they can be safely relaxed if certain criteria are fulfilled, in which case they no longer remain fusion-preventing.

Thus, it is important for the compiler to address these weaknesses to be a 'friend in deed' of the programmer.

1.1.3 Unscalability: The traditional woe of compilers

The third and the most important reason for the poor performance of current production compilers especially for large programs is their own choice in doing so. This choice, is however, forced, because they need to maintain programmer productivity through short compile times. That is, the compilers choose to not consider global program optimizations (or optimizations spanning a large scope of statements) so as to circumvent the issue of analyzing a large number of dependences for reasoning the application of a certain optimization such as fusing multiple loop nests. In short, the compiler's capability of performing useful optimizations does not scale to large programs.

Traditionally, large compile times have been associated with analyzing (many) dependences in a large program to generate the Program Dependence Graph (PDG). The PDG is then used to reason optimizations such as parallelization, fusion, distribution, etc. However, our experiments with a state-of-the-art polyhedral compiler that is armed with the capability of analyzing large scopes and performing useful global optimizations, reveal that the unscalability of compilers is more a problem when applying useful optimizations than just analyzing dependences to build the PDG. For example, it takes less than a second to analyze dependences and construct the PDG in a scientific application, *lu*, whereas effectively optimizing it takes around 3 hours. We are not aware of an existing work that attempts to tackle this important problem, and this problem is in fact, now well recognized in the polyhedral compiler community as a key challenge.

1.2 Contribution of this thesis

This thesis makes contributions to address all three above-mentioned challenges involved in designing parallelizing compilers. In addition to parallelism, we focus on three key compiler optimizations to improve the memory system performance of applications, which in effect amounts to parallel performance. The three optimizations are loop tiling, loop fusion and data prefetching. Through our work on these three optimizations, we address all three challenges as follows.

1. We look at memory optimizations in light of the present-day processors that contain the recent advances in microarchitecture such as multi-level caches, multithreading, vectorization, prefetching. In particular, we revisit the locality enhancing optimization, loop tiling, such that interference misses in caches that stem from the caches being set-associative instead of fully-associative, are minimized, and the chosen tile size best benefits from vectorization and prefetching. Its interaction with multi-threading is also considered, and a generic algorithm that takes these parameters as input is presented, which can thus find utility in any compiler on any host. Similarly, the latency hiding optimization, data prefetching, is visited in light of the hardware supporting prefetching in current processors, a multi-level cache hierarchy, and also multithreading. The decision on the aggressiveness of loop fusion is also made based on cache sizes and maximizing benefits of vectorization. Experimental results indicate significant performance gains over the production compilers on existing multi- and many-core processors when considering the important recent advances in microarchitecture.

2. The solution to the challenge of optimizing large programs includes, (1) relaxing the extra-stringent fusion preventing dependences between temporary variables across loop nests so as to enable effective fusion while also preserving program correctness in the wake of relaxed dependences, and (2) implementing an effective cost model that chooses wise heuristics to decide the statements to be fused to achieve global data reuse; the fusion achieved is such that it does not hurt coarse-grain parallelism. This work is implemented in the state-of-the-art PLuTo polyhedral compiler and has been shown to provide parallel performance improvement of as much 2.17x for individual applications, and as much as

6.8x over the Intel compiler for individual hot regions in those applications when run on an 8-core Intel Xeon processor.

3. The unscalability problem at its very root stems from the large number of program statements and dependences within hot regions in real application programs. The algorithms employed for the purpose of computing transformations have a time and memory complexity that varies as a large power in the number of statements, and thus become unscalable. We address this problem with a one-shot solution - condense the program (or hot region) to be represented by a semantically equivalent but smaller set of statements and dependences. Essentially, we choose a single representative statement in an entire loop, which we call an *Optimization-molecule*. This condensation also then helps us to condense the set of program dependences. With this condensation, global program transformations such as loop fusion (and its supporting transformations such as interchange and shifting) can still be effectively reasoned, and that too, at a much lower overhead. Experimental results indicate significant improvement in compile time and memory requirement for program subroutines with more than 100 statements.

1.3 Organization of the thesis

In the following chapters of this thesis, each compiler optimization is dealt separately with the focus on addressing the above-identified 3 challenges for parallelizing compilers. Chapter 2 provides the background for the thesis - it introduces the three compiler optimizations that are dealt in this work and particularly provides a gentle introduction to the polyhedral compiler framework. Chapter 3 discusses loop tiling and tile size selection for present-day processors, clearly showing the impact of hardware on this optimization and then our solution to account for all of them in a neat algorithm. Chapter 4 discusses data prefetching as an important latency hiding optimization for both the latest multi-core and the many-core processors. This chapter shows how the optimal prefetching strategy on each platform is clearly a function of the host platform and is widely different on these two platforms. Particularly, the best performing strategy on one is the worst on the other and vice-versa. On each platform, we propose an algorithm for the compiler to selectively prefetch data using carefully tuned prefetch distance at different levels of cache, and also coordinate with the existing hardware prefetcher on the host platform when helpful. Chapters 5 through 7 deal with global program optimizations, particularly loop fusion for real application programs. Chapter 5 presents our solution to relaxation of dependences on temporary variables across loop nests to enable effective loop fusion in large programs. Chapter 6 complements Chapter 5 by providing an effective cost model to decide what statements should be fused to benefit most from data reuse while preserving parallelism. Chapter 7 nicely closes this discussion by addressing a long-standing problem in compilers - scalability. Thus, finally, real applications can be effectively optimized for locality and parallelism, and that too, at a cheap price in terms of time and memory. Finally, we present the conclusions from our work in Chapter 8.

1.4 Related Publications

Portions of the work presented in this thesis have been published in the form of three papers. They are as follows.

1. **Sanyam Mehta**, Gautham Beeraka and Pen-Chung Yew. Tile Size Selection Revisited. In ACM Transactions on Architecture and Code Optimization, 10, 4, Article 35 (December 2013), 27 pages.

2. **Sanyam Mehta**, Pei-Hung Lin and Pen-Chung Yew. Revisiting Loop Fusion in the Polyhedral Framework. In Proceedings of the 19th ACM SIGPLAN symposium of Principles and Practice of Parallel Programming (PPoPP '14). ACM, 233-246.

3. **Sanyam Mehta**, Zhenman Fang, Antonia Zhai and Pen-Chung Yew. Multi-stage Coordinated Prefetching for Present-day Processors. In Proceedings of the 28th ACM International Conference on Supercomputing (ICS '14). ACM, 73-82.

Chapter 2

Background

Traditionally, compiler optimizations for improving the memory performance of a processor have been broadly categorized into, (1) *locality enhancing*, and (2) *latency hiding* optimizations.

Many application programs involve multiple updates (writes) to the same data in the form of arrays, lists, etc. Such programs thus *reuse* data. However, the program may be written in a way that it cannot reuse the data in a faster level of memory hierarchy due to its limited size. Thus, *locality enhancing* optimizations transform the program to allow such data reuse. This prevents incurring costly off-chip memory accesses, and also reduces the use of off-chip bandwidth, both amounting to improved performance. *Loop tiling* and *loop fusion* are the two most prominent *locality enhancing* compiler optimizations.

In other applications, there may not be much opportunity to reuse data, or at least the compiler may not be capable of automatically transforming the program to extract such reuse. In such cases, an application (particularly if it is memory intensive as is the case with most scientific applications) could suffer performance loss due to waiting for data to arrive from memory. In such cases, *data prefetching* as a *latency hiding* optimization comes to the rescue. In this optimization, the compiler identifies the data needed in the immediate future, and pro-actively requests for it, so that the execution can proceed unimpeded without suffering from memory latency. Thus, *data prefetching* makes for the slow memory speed, and boosts program performance.

In the rest of this chapter, we provide more specific background for each of these three optimizations. In particular, loop fusion and the polyhedral compiler framework are discussed in more detail.

```
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    for(k = 0;  k< N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

matmul

```
for (iT=0; iT< N/I; iT++)
 for (jT=0; jT< N/J; jT++)
  for (kT=0; kT< N/K; kT++)
   for  (i = I*iT ;  i < I*iT + I-1 ; i++)
    for  (k = K*kT ;  k < K*kT + K-1 ; k++)
     /*  vector loop  */
     for  (j = J*jT ;  j < J*JT + J-1 ; j++)
      C[i][j] += A[i][k] * B[k][j];
```

3D-tiled matmul

Figure 2.1: Matmul - untiled (left) and tiled (right)

2.1 Loop Tiling and Tile Size Selection

Loop tiling converts a program that originally involves sequential traversal of the data (in arrays) to data traversal in tiles. This reduces the amount of data accessed between consecutive accesses to the same (i.e. reusable) data, and thus promotes data reuse in faster and smaller levels of the memory hierarchy such as caches.

Figure 2.1 shows the original *matmul* (matrix-multiplication) kernel and also its tiled version. The 3 loops in the original code become 6 loops, 3 of which traverse the data within the tile and are called the intra-tile loops, and the remaining 3 loops iterate between tiles and are called the inter-tile loops. In the figure, the loops (iT-jT-kT) in the tiled program are the inter-tile loops, and loop (i-j-k) are the intra-tile loops. To aid the understanding of our work on tile size selection in relation to loop tiling, it is important to understand the following concepts/definitions.

There are 2 types of data reuse that are especially relevant to tile size selection:

Self-temporal reuse. This happens when the same reference reuses a data item in distinct loop iterations. For example, the array reference $B[k][j]$ in the tiled *matmul* kernel in Figure 2.1 reuses the same data items in every iteration of loop i, and thus can be said to have *self-temporal reuse* in loop i. Similarly, array references $A[i][k]$ and $C[i][j]$ have self-temporal reuse in loops j and k, respectively. However, array reference $B[k][j]$ reuses an entire tile in each iteration of the outermost loop i, array reference $C[i][j]$ reuses a tile-row in each iteration of loop k, and the array reference $A[i][k]$ reuses only an element in each iteration of loop j.

Thus, only array references with self-temporal reuse in the outermost loop such as $B[k][j]$ provide opportunity for considerable data reuse. It is only in the presence of such references in the source code that loop tiling gives a significant benefit. Thus, it is important to track such references and focus on minimizing conflict misses caused by such references.

Self-spatial reuse. This happens when a reference reuses the same cache line in distinct loop iterations. For example, the array references $C[i][j]$, $A[i][k]$ and $B[k][j]$ in the tiled *matmul* kernel in Figure 2.1 all reuse same cache lines for multiple successive iterations of loops j, k and j, respectively, and can be thus said to have *self-spatial reuse* in the respective dimensions. It is for this reason that self-spatial reuse is best availed when the tile dimensions are a multiple of the cache line size.

Data reuse is enabled by data locality, i.e. data is reused if the data is not replaced from the cache before subsequent use. Thus, corresponding to the 2 types of data reuse, there are 2 types of data locality, i.e. *self-temporal* and *self-spatial* locality. Loop tiling improves the self-temporal locality of data by reducing its reuse distance. Since the reuse distance is a function of the tile size, tile size should be chosen such that data items accessed within a tile are not replaced from the cache due to capacity or conflict misses.

In the tiled code, capacity misses can be easily avoided by choosing a tile whose working set size is smaller than the cache capacity. This, however, doesn't avoid conflict misses that result from non-contiguous memory accesses within the tile. The conflict (or interference) misses are of 2 types - *self* and *cross interference* misses. Self interference misses occur when a reference with self-temporal reuse accesses multiple data items that collide in the cache, i.e. they are mapped to the same set. For example, in tiled *matmul*, self-interference occurs when data items accessed by the reference $B[k][j]$ collide. This is demonstrated in Figure 2.2a which shows the snapshot of a 32KB 4-way set associative L1 cache with 16-element cache lines, at the instance when self-interference begins to cause misses during the execution of an array tile. Self-interference is pronounced for problem sizes[1] that are a power of 2. This is because the cache size is also a power of 2. In such a case, different rows in the tile are prone to map to the same set. This is seen in Figure 2.2a where the leading dimension of array B (ld_B) is 512 - the 17^{th} row of the tile interferes with the first row of the tile. Thus, for the given cache configuration and problem size, the tile height, or number of rows in the tile, should not exceed 16 or else conflict misses will result. In this example, the tile width was chosen to be 16, the size of the cache line, but the case of pronounced conflict misses depends primarily on the tile

[1]Problem size refers to the size of the array or matrix in a program. However, in this chapter, we mention problem size to particularly refer to the *leading dimension* of the array, which is critical for tile size selection. For row-major 2D arrays as in C, elements from one row to the next are a *leading dimension* apart.

height.

Cross interference misses occur when two or more references collide in the cache and one of them has self-temporal reuse. For example, in the tiled *matmul* kernel, cross-interference occurs when data items accessed by the references to arrays A, B and C collide. Figure 2.2b shows a snapshot of the cache demonstrating cross interference between references to arrays A and B, where the leading dimension of the arrays is 2000. In the figure, we assume that references A[0][0] and B[0][0] map to sets 0 and 6, respectively. However, run-time addresses of the arrays are not known at compile-time and thus cross interference is harder to account for. Among works based on an analytical model, (3) and (4) have attempted to minimize cross-interference misses, but only for direct-mapped caches.

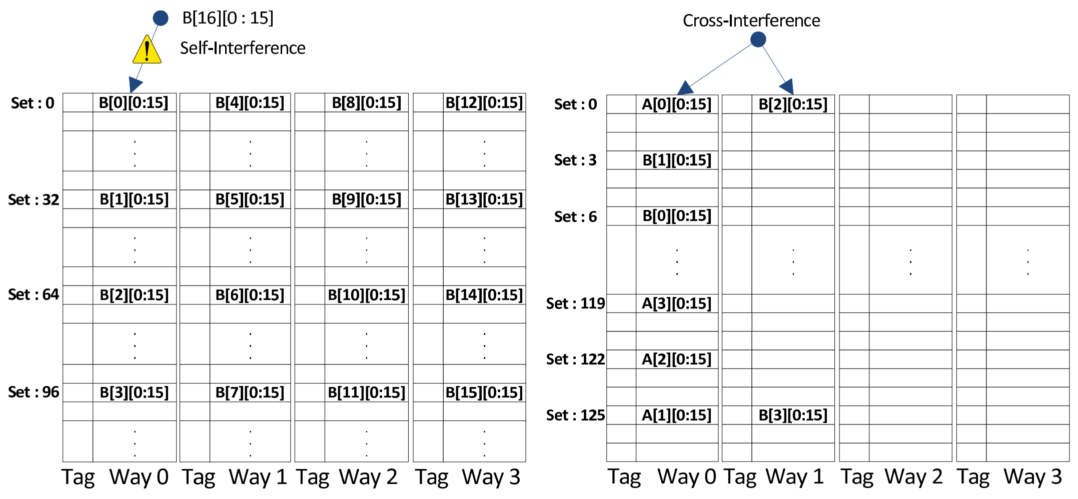


Figure 2.2: (a) Demonstration of self-interference ($ld_B = 512$); (b) Demonstration of cross-interference ($ld_A = ld_B = 2000$); [ld_A and ld_B are leading dimensions of arrays A and B, respectively]

2.2 Data Prefetching

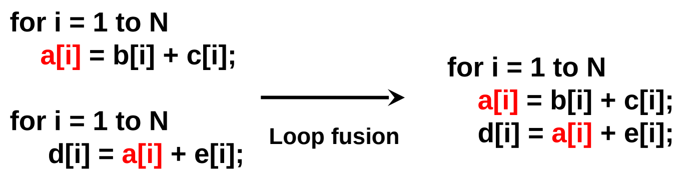
Data prefetching involves detection of specific access patterns in the executing program and leveraging it to issue requests for data to be accessed in the future to hide the latency of data access. For example, for the (untiled) *matmul* kernel shown in Figure 2.1, we see that consecutive elements of array A will be referenced since loop k is the innermost loop, and it corresponds to its fastest running subscript. Thus, future data needed by such a reference is easily predicted, and prefetched for performance improvement. However, *matmul* is not particularly memory intensive due to available reuse, and thus benefits more from tiling. But, not all applications have such inherent reuse. Over the years, data prefetching has proved to be very useful for memory intensive applications, and thus both hardware and software techniques exist to detect access patterns and thereby issue prefetch requests.

In *hardware-based prefetching*, some special hardware monitors data access patterns to a particular cache and identifies data suitable for prefetching based on obtained information. Current processors employ multiple hardware prefetchers for *streaming* as well as *strided* accesses.

Software-directed prefetching, on the other hand, involves the insertion of prefetch instructions into the original code by the programmer or the compiler that request data needed a few iterations later. This distance in the number of loop iterations is called the *prefetch distance*. Like hardware prefetchers that sit on multiple levels of cache and can prefetch data to those levels, the latest instruction sets provide prefetch intrinsics for prefetching data at different levels of cache.

In software-directed prefetching, it is the responsibility of the programmer to ensure timeliness and prevent redundant prefetches by deciding the data to prefetch and the prefetch distance. The hardware prefetchers usually ensure timeliness through aggressive prefetching, i.e. maintaining a prefetch degree of more than one. For example, the streamer hardware prefetchers on SandyBridge (a multi-core processor) and Xeon Phi (a many-core processor) have prefetch degrees of 2 and 4, respectively, and can maintain a prefetch distance of a maximum of 20 cache lines. Depending on implementation, software prefetch instructions can also be used to train and thus control the prefetch distance at which the hardware prefetcher operates. *Comparing software-directed and hardware-based prefetching, software-directed prefetching has the advantage of being used in a controlled manner, but is associated with an additional instruction overhead which may compete with the gains.*

While the above-mentioned facts about prefetching are better known, the impact of other hardware features influencing prefetching are less well understood. We brief this interaction between hardware features and prefetching here to help the understanding of our strategy of coordinated prefetching in Chapter 4. The hardware tracks the outstanding prefetch requests through a buffer or a queue. This hardware structure is called the *Line Fill Buffer* (LFB) in SandyBridge and MSHR (*Miss Status Handling Registers*) file in Xeon Phi, and is responsible for rendering the data prefetch requests *non-blocking*. The size of the LFB or the MSHR file,



```
for i = 1 to N
    a[i] = b[i] + c[i];

for i = 1 to N
    d[i] = a[i] + e[i];
```

```
for i = 1 to N
    a[i] = b[i] + c[i];
    d[i] = a[i] + e[i];
```

Figure 2.3: Loop fusion

among other factors, has a significant bearing on the most appropriate choice of prefetching strategy on a particular architecture. This is because, on Xeon Phi, if the MSHR file is full, the pipeline stalls. On SandyBridge, if the LFB is full, subsequent prefetches/loads enter the load buffer, which when full, stalls the pipeline. Thus, any prefetching strategy must issue prefetch requests in such a way that it leads to minimum contention for this scarce resource - the LFB or the MSHR file.

2.3 Loop Fusion and Polyhedral Compiler Framework

Loop fusion groups references to the same data by merging the loop bodies of multiple loop nests into the same nest. For example, in Figure 2.3, the array a is referenced in two different loop nests which leads to access to similar data in both nests. In the fused program, the two references are grouped into the same nest. As a result, the successive uses of the same data (of array a) happen in the same iteration of loop i, as compared to the original program, when such accesses were separated by N loop iterations. This improved temporal locality of data reduces the costly accesses to off-chip memory, amounting to improved performance.

In this dissertation, we implement loop fusion within the polyhedral compiler framework, and use our framework to then address loop fusion in real applications. Thus, we next provide a gentle introduction to the polyhedral framework with particular emphasis on aspects relevant to our work in this thesis. The reader is referred to existing literature (5; 6; 7) for more detail on polyhedral compiler frameworks.

DEFINITION 1 (Affine Hyperplane). An affine hyperplane is an n - 1 dimensional affine sub-space of an n dimensional space. An affine hyperplane can be viewed as a one-dimensional affine function that maps an n-dimensional space onto a one-dimensional space, or partitions an n-dimensional space into n-1 dimensional slices. Hence, as a function, it can be written as, $\Phi(\vec{v}) = \text{h}.\vec{v} + c$. A hyperplane divides the space into two half-spaces, the positive half-space, and a negative half space.

DEFINITION 2 (Polyhedron). A polyhedron is an intersection of a finite number of half-spaces. A polytope is a bounded polyhedron.

LEMMA 1 (Affine form of the Farkas lemma). If a non-empty polyhedron is defined by p inequalities or faces,

$$\mathbf{a_k}\vec{x} + b_k \geq 0, k = 1, p \tag{2.1}$$

then, an affine form ψ is non-negative everywhere in that polyhedron iff it is a non-negative linear combination of the faces:

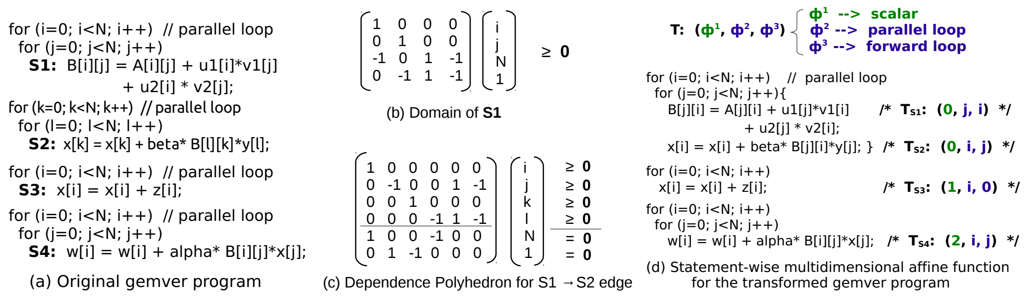
$$\psi(\vec{x}) \equiv \lambda_0 + \sum_{k=1}^{p} \lambda_k(\mathbf{a_k}\vec{x} + b_k), \lambda_0, \lambda_1, ..., \lambda_p \geq 0 \tag{2.2}$$

2.3.1 Overview of the Polyhedral Framework

The polyhedral framework for compiler optimizations is a powerful mathematical framework based on parametric linear algebra and integer linear programming. It provides an effective intermediate representation that captures nearly all the complex high level optimizations performed by a traditional automatic parallelizing compiler.

The polyhedral framework performs loop transformations on a *Static Control Part* (SCoP), a maximal set of consecutive statements $(S_1, S_2, ..., S_n)$, where loop bounds and conditionals are affine functions of the surrounding loop iterators and parameters. The *iteration domain* of the statements within a SCoP can be specified as a set of linear inequalities in the polyhedral framework as shown in Figure 2.4b. This set of linear inequalities defines a polyhedron, with each iteration of a loop represented by an integer point within this polyhedron. With such an abstraction, it is possible not only to obtain the exact dependences between statements, but also to model a composition of complex transformations as a single algebraic operation.

A dependence between two statement instances belonging to statements S_i and S_j respectively, is represented by a set of equalities and inequalities in the *dependence polyhedron*, $P_{e^{S_i \rightarrow S_j}}$, where $e^{S_i \rightarrow S_j} \in E$ is an edge in the Data Dependence Graph (DDG), $G = (V, E)$, with each vertex representing a statement. The dependence polyhedron not only captures the iteration domain of the statements involved in the dependence, but also the exact dependence



```
for (i=0; i<N; i++)  // parallel loop
  for (j=0; j<N; j++)
    S1:  B[i][j] = A[i][j] + u1[i]*v1[j]
                   + u2[i] * v2[j];
for (k=0; k<N; k++)  // parallel loop
  for (l=0; l<N; l++)
    S2:  x[k] = x[k] + beta* B[l][k]*y[l];
for (i=0; i<N; i++)  // parallel loop
  S3:  x[i] = x[i] + z[i];
for (i=0; i<N; i++)  // parallel loop
  for (j=0; j<N; j++)
    S4:  w[i] = w[i] + alpha* B[i][j]*x[j];
```

(a) Original gemver program

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq \mathbf{0}$$

(b) Domain of **S1**

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \\ \hline 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ l \\ N \\ 1 \end{pmatrix} \begin{array}{c} \geq \mathbf{0} \\ \geq \mathbf{0} \\ \geq \mathbf{0} \\ \geq \mathbf{0} \\ \hline = \mathbf{0} \\ = \mathbf{0} \end{array}$$

(c) Dependence Polyhedron for S1 → S2 edge

$$\textbf{T:}\ (\phi^1, \phi^2, \phi^3) \begin{cases} \phi^1 \dashrightarrow \textbf{scalar} \\ \phi^2 \dashrightarrow \textbf{parallel loop} \\ \phi^3 \dashrightarrow \textbf{forward loop} \end{cases}$$

```
for (i=0; i<N; i++)    //  parallel loop
  for (j=0; j<N; j++){
    B[j][i] = A[j][i] + u1[j]*v1[i]        /*  TS1:  (0, j, i)  */
                  + u2[j] * v2[i];
    x[i] = x[i] + beta* B[j][i]*y[j]; }  /*  TS2:  (0, i, j)  */
for (i=0; i<N; i++)
   x[i] = x[i] + z[i];                    /*  TS3:  (1, i, 0)  */
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    w[i] = w[i] + alpha* B[i][j]*x[j];   /*  TS4:  (2, i, j)  */
```

(d) Statement-wise multidimensional affine function for the transformed gemver program

Figure 2.4: Overview of the Polyhedral Framework

information such as which iterations are involved in the dependence. For example, in Figure 2.4c, the equalities express that there is dependence when i=l and j=k. The dependence polyhedron could express such a precise dependence relation because there existed an affine relation between the iterations and the referenced data.

Since the dependence polyhedron captures dependences among all loop iterations for affine programs, the dependence information in the DDG is *exact*. With this exact dependence information, the goal is to find a statement-wise multi-dimensional affine function (T) to represent a composition of loop transformations for the entire SCoP. Each dimension or level of this multi-dimensional affine function is represented by $\phi(\vec{i})$ and is defined as follows:

$$\phi_S(\vec{i_S}) = (c_S^1\ c_S^2\ ...\ c_S^{m_S})(\vec{i_S}) + c_S^0 \tag{2.3}$$

where $\vec{i_S}$ is the loop iteration vector for statement S, and $c_S^0 ... c_S^{m_S}$ are constants. Figure 2.4d shows the multi-dimensional affine functions (T_{S1} through T_{S4}) for each of the four statements of the *gemver* benchmark, where each function has 3 dimensions or levels represented by ϕ^1, ϕ^2, and ϕ^3, respectively.

The one-dimensional affine transform (ϕ) for each statement can either specify a *loop hyperplane* or a *scalar dimension*. A *loop hyperplane* is an $n-1$ dimensional sub-space of an n dimensional space represented by the normal $\left(c_S^1\ c_S^2\ ...\ c_S^{m_S}\right) \neq \vec{0}$. A legal loop hyperplane corresponds to a loop in the transformed program. For example, for *gemver*, $\phi_{S2}^2 = (1\ \ 0)(i\ \ j)^T = i$, and it represents a hyperplane that corresponds to the outermost loop (i-loop) for statement S2 as shown in Figure 2.4d. A legal hyperplane does not violate any unsatisfied dependence, $e^{S_i \rightarrow S_j}$, at that loop level, i.e.

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) \geq 0, \langle \vec{s}, \vec{t} \rangle \in P_{e^{S_i \to S_j}} \tag{2.4}$$

where $\vec{s}$ (source) and $\vec{t}$ (target) are instances of statements S_i and S_j, respectively. The above condition implies the loop hyperplane preserves the direction of dependences between any two instances of statements S_i and S_j.

The above condition (2.4), when expanded becomes,

$$\left(c^1_{S_j}\ c^2_{S_j}\ ...\ c^{m_{S_j}}_{S_j}\right)\vec{t} - \left(c^1_{S_i}\ c^2_{S_i}\ ...\ c^{m_{S_i}}_{S_i}\right)\vec{s} \geq 0, \langle \vec{s}, \vec{t} \rangle \in P_{e^{S_i \to S_j}} \tag{2.5}$$

This is, however, non-linear in the unknown co-efficients of the phis and loop index variables. Thus, the affine form of the Farkas lemma is used for linearizing this legality condition as follows.

$$\left(c^1_{S_j}\ c^2_{S_j}\ ...\ c^{m_{S_j}}_{S_j}\right)\vec{t} - \left(c^1_{S_i}\ c^2_{S_i}\ ...\ c^{m_{S_i}}_{S_i}\right)\vec{s} \equiv \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek} P^k_e, \lambda_{ek} \geq 0 \tag{2.6}$$

Thus, the non-linear form in terms of the loop variables (Equation 2.5) is now expressed as a non-negative linear combination of the faces of the dependence polyhedron through the use of Farkas lemma. Now, the coefficients (denoting a hyperplane) on the LHS and RHS can be equated to get rid of the loop variables. It is important to note that the legality condition has to be satisfied for every dependence in the SCoP, and thus Farkas lemma is also applied for every dependence. The resulting constraints (linear in the coefficients of the phis) are aggregated. For the purpose of eliminating the Farkas multipliers, Fourier-Motzkin elimination is employed.

In the polyhedral model, a fusion partitioning can be represented by a *scalar dimension* in the multi-dimensional affine function. For a scalar dimension, $c^1_S = c^2_S = ... = c^{m_S}_S = 0$, $\forall$ S, and c^0_S determines the partition number that the statement belongs to. The statements with the same value of c^0_S belong to the same partition, or in other words, they are fused at that loop level. For example, for *gemver*, ϕ^1 represents a scalar dimension, and statements S1 and S2 both have $c^0_S = 0$, and as a result, they are perfectly fused in the transformed *gemver* code shown in Figure 2.4d. Since, c^0_S is 1 and 2, respectively, for statements S3 and S4, they are distributed in the transformed code.

Thus, a multi-dimensional affine transform consists of multiple one-dimensional affine transforms representing legal loop hyperplanes interspersed by scalar dimensions. This multi-dimensional affine transform, because of the *convex combination* property, allows it to capture a sequence of simpler transformations such as coarse-grained parallelism, loop interchange, skewing, shifting, fusion and tiling. For example, in *gemver*, it allowed to capture parallelism, interchange and fusion in a single optimization step as seen in Figure 2.4d.

Chapter 3

Tile Size Selection, from Then to Now

3.1 Introduction

Loop tiling (8; 9; 10; 11) is a widely used loop transformation to enhance data reuse in higher levels of memory hierarchy. In essence, loop tiling reduces the *reuse distance* from being a function of the problem size to a function of the tile size. Loop tiling thus minimizes the *cache capacity* misses. However, tiling leads to non-contiguous data accesses in memory resulting in an increased probability of *conflict misses* in the cache. These conflict misses are a function of the tile size in tiled code. For example, for a problem size of (N=2000), a 2D tiled code of the *dsyr2k* kernel from BLAS library (12) with the tile size (TS_1=8x128) performs 2.44x better than the one with the tile size (TS_2=128x8) even though both codes have the same working set size in the L1 cache, when tested on an Intel Xeon processor based on the Sandy Bridge microarchitecture. One of the key reasons for this significant difference in performance is that the tile size TS_2 leads to pronounced conflict misses, that are 22 times more compared to those observed for tile size TS_1.

Thus, tile size selection is critical to performance of the tiled code, and an optimal tile size is one which minimizes not only capacity but also conflict misses within a tile.

In the past, the problem of tile size selection has been attempted using analytical model-driven approaches (13; 14; 3; 15; 16; 4; 17; 18). However, *these approaches have proved to be less robust because the models used did not fully capture the interaction between the source program (features like problem size and reuse characteristics of the arrays) and critical features in the modern processor microarchitecture such as multi-level and set-associative caches, and the*

SIMD unit. This has resulted in a widening gap between performance delivered by best known tile sizes and that achieved by using tile sizes predicted by the previous analytical models.

Tile size selection has also been widely studied using empirical auto-tuning as in ATLAS and PHiPAC for linear algebra (19; 20), FFTW and SPIRAL for Signal Processing (21; 22), ETile (23), and others (24; 25). However, each of these techniques is faced with a large search space of tile sizes when considering multidimensional, non-cubic tiling. For example, the size of search space for a level-3 BLAS kernel such as *gemm* scales at n^3, where n is a function of the size of cache being considered. If reuse opportunity in multiple levels of cache must be considered, the search space expands significantly for the larger caches, lower in the memory hierarchy. As a result, the adopted approach is either too time consuming (19; 21; 20; 22; 24) or less accurate when heuristics are used to reduce the search space (25).

In this work, we propose a new analytical model for selecting tile size in modern processors. The *Tile Size Selection (TSS) algorithm* proposed in our model chooses a tile size such that the following 4 objectives are met,

- it leverages the high set-associativity in modern caches to minimize interference and yield stable performance for all ranges of problem sizes in a variety of source programs.
- it considers data reuse at multiple levels of cache, which further boosts the performance of the tiled code.
- it considers the interaction of tiling with the SIMD unit on the host architecture and chooses a tile size that best benefits from it.
- it considers the impact of tiled code execution in a multithreaded environment (both chip multiprocessing and simultaneous multithreading environments) and achieves good performance.

Although the analytical model developed in this work chooses a particular tile size, it could also be used in conjunction with an auto-tuning framework to prune/navigate the search space.

We implement the TSS algorithm within a source-to-source polyhedral compiler framework, PLuTo (26), which automatically tiles the source programs. For the generated tiled code, PLuTo chooses a cubic tile size by default. The TSS algorithm instead, estimates the optimal tile size which is then used by PLuTo to generate the corresponding tiled code. We tested our model on 12 benchmarks comprising a mix of linear algebra, data-mining and stencil kernels, all of which are known to benefit significantly from tiling. We tested each kernel with 2 different problem sizes on two different machines with different microarchitectures. Experimental results show that accommodating the impact of multi-level caches and the SIMD unit within the analytical model adds significant performance to the tiled code, and the tile size thus chosen is similar to that obtained from an exhaustive search for the best tiled code in a constrained search space. In comparison to the best square (cubic) tiled code for the test benchmarks, our tile size selection algorithm chooses tile sizes that perform 9.7% and 20.4% faster on average, respectively, for the 2 problem sizes. We also show that our model achieves good performance for tiled codes run in multithreaded environment using either the *chip multiprocessing* or the *simultaneous multithreading* technology.

The rest of this chapter is organized as follows. Section 3.2 re-evaluates the problem of tile size selection from the point of view of a compiler via a case study and reinforces the motivation for this work. Section 3.3 discusses the multiple factors that influence tile size selection and explains our approach to accommodate them in our tile size selection model. In Section 3.4, we describe our algorithm for estimating the optimal tile size and discuss its scope. Section 3.5 describes the experimental setup and the benchmarks used for our experiments. We discuss the experimental results in Section 3.6. The related work is presented in Section 3.7. Finally, we conclude in Section 3.8.

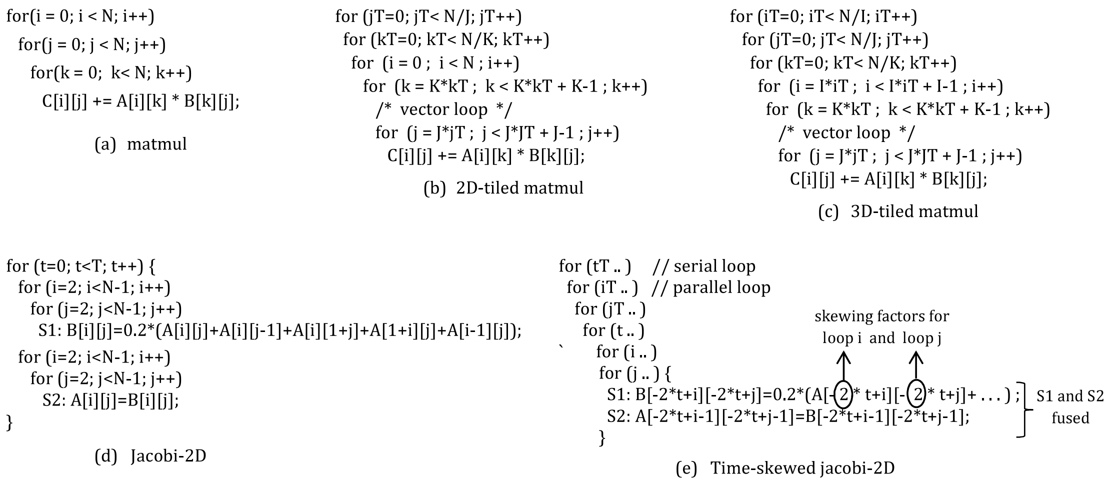
3.2 Motivation

Tile Size Selection has been deemed a complex problem from the point of view of a compiler. In this section, we re-evaluate the major reasons behind such a view and present a viable alternative to tackle the problem of tile size selection at compile time.

3.2.1 Modeling the effect of set-associativity.

An analytical model for tile size selection must choose a tile size such that the conflict misses are minimized. Past works did not consider set-associative caches and proposed probabilistic approaches (3; 4; 27; 28; 29) for minimizing conflicts among array references. In such a scenario, a conservative approach based on eliminating conflict misses as in (3; 4) results in tiles with undesirable shapes (too “skinny” or too “fat” tiles) for some problem sizes, leading to sub-optimal performance in those cases. On the other hand, an optimistic approach as in (27; 28; 29) is based on ignoring the non-unity set associativity of caches. This too gives below optimal performance because multiple cache lines accessed in the source program may map to the same set of an n-way set associative cache, causing conflicts more likely than that predicted by such an optimistic analytical model.

With multiple array references within the loop body of a source program, modeling the effect of set associativity on the execution time behavior of a tiled code has been considered non-trivial. We re-evaluate this problem in the example of matrix-multiplication (*matmul*) kernel shown in Figure 3.1.



```
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    for(k = 0;  k< N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

(a) matmul

```
for (jT=0; jT< N/J; jT++)
 for (kT=0; kT< N/K; kT++)
  for  (i = 0 ;  i < N ; i++)
   for  (k = K*kT ;  k < K*kT + K-1 ; k++)
    /*  vector loop  */
    for  (j = J*jT ;  j < J*JT + J-1 ; j++)
     C[i][j] += A[i][k] * B[k][j];
```

(b) 2D-tiled matmul

```
for (iT=0; iT< N/I; iT++)
 for (jT=0; jT< N/J; jT++)
  for (kT=0; kT< N/K; kT++)
   for  (i = I*iT ;  i < I*iT + I-1 ; i++)
    for  (k = K*kT ;  k < K*kT + K-1 ; k++)
     /*  vector loop  */
     for  (j = J*jT ;  j < J*JT + J-1 ; j++)
      C[i][j] += A[i][k] * B[k][j];
```

(c) 3D-tiled matmul

```
for (t=0; t<T; t++) {
  for (i=2; i<N-1; i++)
    for (j=2; j<N-1; j++)
      S1: B[i][j]=0.2*(A[i][j]+A[i][j-1]+A[i][1+j]+A[1+i][j]+A[i-1][j]);
  for (i=2; i<N-1; i++)
    for (j=2; j<N-1; j++)
      S2: A[i][j]=B[i][j];
}
```

(d) Jacobi-2D

```
for (tT .. )    // serial loop
 for (iT .. )  // parallel loop
  for (jT .. )
   for (t .. )
`    for (i .. )
      for (j .. ) {
       S1: B[-2*t+i][-2*t+j]=0.2*(A[-2* t+i][-2* t+j]+ . . . ) ;
       S2: A[-2*t+i-1][-2*t+j-1]=B[-2*t+i-1][-2*t+j-1];
      }
```

(e) Time-skewed jacobi-2D

Figure 3.1: Source codes of *matmul* and *jacobi-2d*, before and after tiling through PLuTo

In *2D-tiled matmul*, the maximum working set that should fit the cache comprises all the data accessed in a single iteration of outermost untiled loop i as shown in Figure 3.1b. Thus, the working set consists of a tile row each in arrays A and C (i.e. K and J elements, respectively), and an entire tile of array B (i.e. K*J elements)[1]. Clearly, the working set is dominated by the data accessed by array reference B. Further, if this working set fits a particular level of cache, the entire tile of array B can be reused in each iteration of loop i. Thus, for *matmul* (and similar other kernels that allow data reuse), the task of modeling the effect of set-associativity for multiple array references in the source program is reduced to only those references with temporal reuse in the outermost loop such as the array reference B. Therefore, in a set-associative cache, most ways could be assigned to such a reference and the effect of set-associativity be modeled only for that reference, independent of the other array references. However, if there are multiple array references with temporal reuse in the outermost loop (as in *jacobi-2d* kernel shown in Figure 3.1d that has 2 such references, $A[i][j]$ and $B[i][j]$), the available ways in each set can be equally partitioned among them and the effect of set-associativity be modeled for any one of

[1]If an LRU cache replacement policy is considered instead of an optimal replacement policy, then the working set will also include another tile row of array C and an element of array A (30)

them as all such references exhibit similar runtime behavior.

3.2.2 Tapping into data reuse at multiple levels of cache.

With multi-level caches on modern processor architectures, tapping into the reuse opportunities at the different levels of cache is important for effective tiling. For example, the best 2D tile for *matmul* performs 13% worse than the best 3D tile that considers data reuse in the L2 cache on an Intel Sandy Bridge core for a problem size of 2000. *The possibility of reusing data in different levels of cache arises from the fact that different array references have different reuse distances*. For example, in the *3D-tiled matmul* shown in Figure 3.1c, each tile of array B is reusable in every iteration of loop i, whereas each tile of array C is reusable in every iteration of loop kT.

While reusing data in multiple levels of cache is important, past works have assumed 2D tiling and determine tile sizes that target data reuse in a single level of cache, generally the L1 cache. If data reuse in caches that are lower in the memory hierarchy (such as L2, L3 caches) must be considered, the impact of set-associativity assumes greater significance as those caches usually have higher set-associativity than the L1 cache. Hence, ignoring set-associativity will lead to unoptimal cache utilization and limited performance gains.

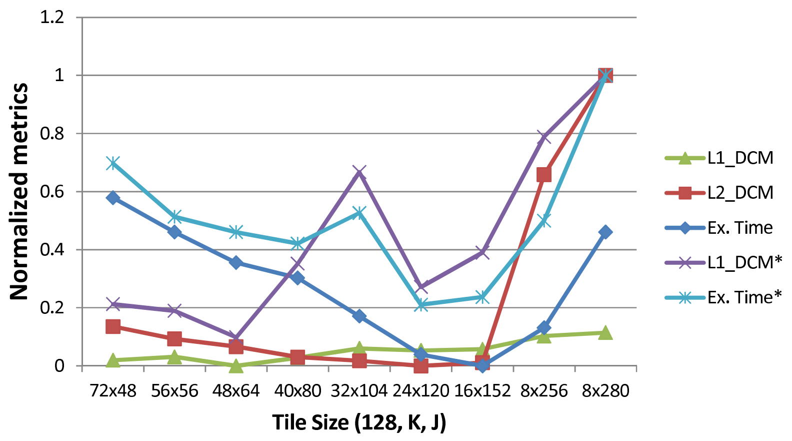


Figure 3.2: Normalized metrics for *matmul* (N=2000); L1 size = 32KB; L2 size = 256KB

Figure 3.2 substantiates the various arguments made in this section to motivate our work. Figure 3.2 plots 3 different metrics, execution time, L2 data cache misses and L1 data cache misses, for the *matmul* benchmark tiled in 3 dimensions with 9 different tile sizes. The experiments were all performed on an Intel Xeon processor (E5-2650, 2.0 GHz) with 32KB private L1 cache, 256KB private L2 cache and a 20MB L3 cache shared by 8 Sandy Bridge cores. In each tiled code, the outermost dimension was fixed at 128 to illustrate the performance impact of L2 cache misses. For each metric, the figure plots values normalized within the range 0...1. Normalized values for each metric are computed using the standard min-max normalization as the ratio, (x-min)/(max-min), where x, min and max are the absolute, minimum and the maximum values, respectively, of that particular metric. Each tiled code uses values of J and K such that the tile maximally occupies the L1 cache without incurring conflict misses. The metric *L1_DCM** shows the L1 cache misses for tile sizes that are greater than the 9 sizes shown on the x-axis by just 1 cache line in the K dimension. From the figure the following messages can be taken.

1. The working set for each of the 9 maximal tiled codes is nearly 24 KB[2], which is well within the L1 cache size of 32KB. A small increase in tile size leads to a significant increase in the L1 cache misses as shown by the metric *L1_DCM** in the figure. This increase in the L1 cache misses is attributed to the conflicts that occur in L1 cache even though the cache capacity is not reached. The increase in L1 cache conflict misses is reflected in the increased execution time as compared to that of the maximal tiles as shown in the figure through the metric *Ex. Time**. Thus, it is critical to consider the impact of set-associativity when determining the optimal tile size.
2. With the L1 misses held nearly constant for the 9 maximal tiled codes as shown by the metric *L1_DCM* in the figure, the execution time follows the pattern of L2 cache misses, with a sharp increase in the execution time beyond $J = 152$. This is because, beyond this point, the working set comprising the data accessed within each iteration of the kT loop overflows the L2 cache, leading to loss of reuse of data accessed by reference C. Thus, it is critical to choose a tile size that targets reuse of data in the L2 cache in addition to the L1 cache. A similar analysis was used to motivate the need to consider data reuse in multiple levels of memory hierarchy in (18).

The algorithm presented emulates execution-time behavior of references with reuse in a particular level of cache by considering the *cache size*, *line size* and *set-associativity*. The algorithm considers data reuse in multiple levels of cache to provide an estimate of the optimal

[2]The size of the working set in the L1 cache, as derived earlier in Section 3.2.1, is given by (K*J + 2*J + K + 1)

tile size. The details of our algorithm are presented in Sections 3.3 and 3.4.

3.3 Our Approach

In this section, we discuss the various factors that influence tile size, and our approach for accommodating them in our algorithm. For this purpose, we first consider linear algebra and data mining kernels that have considerable reuse and thus benefit from tiling. The factors affecting tile size selection in such kernels are considered in Sections 3.3.1 through 3.3.4 through the example of *matmul* introduced in Section 3.2. The conclusions presented are, however, applicable to a variety of kernel programs that allow data reuse and thus benefit significantly from the tiling transformation. Although the example of *matmul* that has a 3D loop nest is considered, the approach is applicable to nD loop nests as described in Section 3.4.1. It must be noted that we assume single-level nD tiling (i.e. n inter-tile loops and n intra-tile loops) for a source code that has n loops and still achieve data reuse in multiple levels of cache - *the array reference(s) that has reuse in the outermost intra-tile loop is reused in the L1 cache and the array reference(s) that has reuse in the innermost inter-tile loop is reused in the L2 cache*. In other words, we do not consider multi-level tiling to achieve data reuse in multiple levels of cache.

Apart from linear algebra and data mining kernels, stencils are another class of codes that benefit significantly from tiling. In stencils, tiling is performed after loop skewing, and is called *time skewing* (31). The factors affecting tile size selection in stencils is considered separately in Section 3.3.5.

3.3.1 Data reuse in L1 cache.

Data reuse in L1 cache, that is the fastest among all levels of cache, is critical to performance of a tiled code. For the *3D-tiled matmul* shown in Figure 3.1c, data reuse in the L1 cache is achieved by reusing the data accessed within a tile of array B in every iteration of the outermost intra-tile loop i. This requires the tile dimensions (K, J) to be chosen such that data locality for array reference B is ensured between successive iterations of loop i. We define two metrics to analyze the impact of tile size selection on data reuse:

(1) **Total Cache Misses (TCM)** - The total number of misses incurred in a particular level of cache for the entire execution of the program.

(2) **Reuse Ratio (RR)** - The ratio of the amount of reusable data to the total working set size for a particular level of cache.

TCM for L1 cache as a function of the tile dimensions (I, J, K) is calculated as follows:

The number of cold misses incurred in the first iteration of loop i equals the number of cache lines accessed in each iteration of loop i, i.e. a tile ($J * K$ elements) of array B, and a tile row each of arrays A and C (K and J elements, respectively). It is given by

$$K * \lceil \frac{J}{CLS} \rceil + \lceil \frac{K}{CLS} \rceil + \lceil \frac{J}{CLS} \rceil \tag{3.1}$$

where *CLS* is the cache line size
Since our goal is to reuse a tile of array B in each iteration of loop i, we assume that the tile size is such that there are no capacity or conflict misses that cause the data accessed within a tile of array B to be evicted from the cache. Thus, the total number of cache misses for the I iterations of loop i equals

$$K * \lceil \frac{J}{CLS} \rceil + I * \lceil \frac{K}{CLS} \rceil + I * \lceil \frac{J}{CLS} \rceil \tag{3.2}$$

Thus, TCM for the L1 cache equals

$$\left(K * \lceil \frac{J}{CLS} \rceil + I * \lceil \frac{K}{CLS} \rceil + I * \lceil \frac{J}{CLS} \rceil \right) * \frac{N^3}{I * J * K} \tag{3.3}$$

where $\frac{N^3}{I*J*K}$ are the number of loop iterations in the inter-tile loops. This, when simplified (assuming I, J and K are multiples of CLS) becomes

$$\left(\frac{1}{I} + \frac{1}{J} + \frac{1}{K} \right) * \frac{N^3}{CLS} \tag{3.4}$$

RR for the L1 cache is defined as the ratio of the reusable data (i.e. $J * K$ elements) to the total working set size at L1 cache. The total working set size at L1 cache can be calculated as the sum of *Minimum Working Set Lines* (18) and the correction for the cache replacement policy (as discussed in Section 3.2). It is given by

$$K * J + 2 * J + K + 1 \tag{3.5}$$

Thus, RR for L1 cache is

$$\frac{K * J}{K * J + 2 * J + K + 1} \tag{3.6}$$

From Equations 3.4 and 3.6, two inferences can be drawn -

1. Each of the tile dimensions should be as large as possible for effective data reuse and minimization of cache misses, and
2. RR for the L1 cache is close to 1. This implies that the array reference B dominates the working set, and would occupy most of the ways in a set-associative cache.

3.3.2 Data reuse in L2 cache.

Data reuse in the L2 cache is critical as L2 misses are more costly than L1 misses. Data reuse in L2 cache is achieved by reusing the data accessed within a tile of array C in every iteration of the innermost inter-tile loop kT. We again consider the TCM and RR for the L2 cache to analyze the impact of tile size selection on data reuse. TCM for L2 cache as a function of the tile dimensions, can be calculated as follows.

The number of cold misses incurred in the first iteration of loop kT equals the number of cache lines accessed in each iteration of loop kT, i.e. a tile each of arrays A, B and C ($I * K$, $K * J$ and $I * J$ elements respectively).

$$I * \lceil \frac{K}{CLS} \rceil + K * \lceil \frac{J}{CLS} \rceil + I * \lceil \frac{J}{CLS} \rceil \tag{3.7}$$

Since our goal is to reuse a tile of array C in every iteration of loop kT, we assume that the tile size is chosen such that there are no capacity or conflict misses that cause the data accessed within a tile of array C to be evicted from the cache. Thus, the total number of cache misses for the $\frac{N}{K}$ iterations of loop kT equals

$$\frac{I * N + J * N + I * J}{CLS} * \frac{N^2}{I * J} \tag{3.8}$$

which when simplified becomes

$$\left(\frac{1}{J} + \frac{1}{I} + \frac{1}{N}\right) * \frac{N^3}{CLS} \tag{3.9}$$

RR for the L2 cache is defined as the ratio of the reusable data (i.e. $I * J$ elements of array C) to the total working set size at L2 cache. The total working set size at L2 cache can be calculated as the sum of number of *Distinct Lines* (32) accessed within the intra-tile loops and the correction for the cache replacement policy. This allows L2 reuse in the innermost inter-tile loop. The working set at L2 cache is given by

$$(I + 1) * K + 2 * K * J + I * J \tag{3.10}$$

Thus, RR for L2 cache is

$$\frac{I * J}{(I + 1) * K + 2 * K * J + I * J} \tag{3.11}$$

From Equations 3.9 and 3.11, two inferences can be drawn -

1. The tile dimensions I and J should be as large as possible for effective data reuse and minimization of L2 cache misses, and
2. Unlike L1 cache, the value of RR for the L2 cache cannot be easily estimated.

3.3.3 Data reuse in both L1 and L2 cache

The Tile Size Selection (TSS) algorithm presented in this chapter emulates execution-time behavior of a single reference in a particular level of cache by considering the size of the cache, its line size and associativity. The algorithm thus aims to minimize conflict misses at that level of cache to enable effective data reuse. For the L1 cache, clearly, the reference(s) with temporal reuse in the outermost intra-tile loop such as the array reference B in *matmul*, dominates the

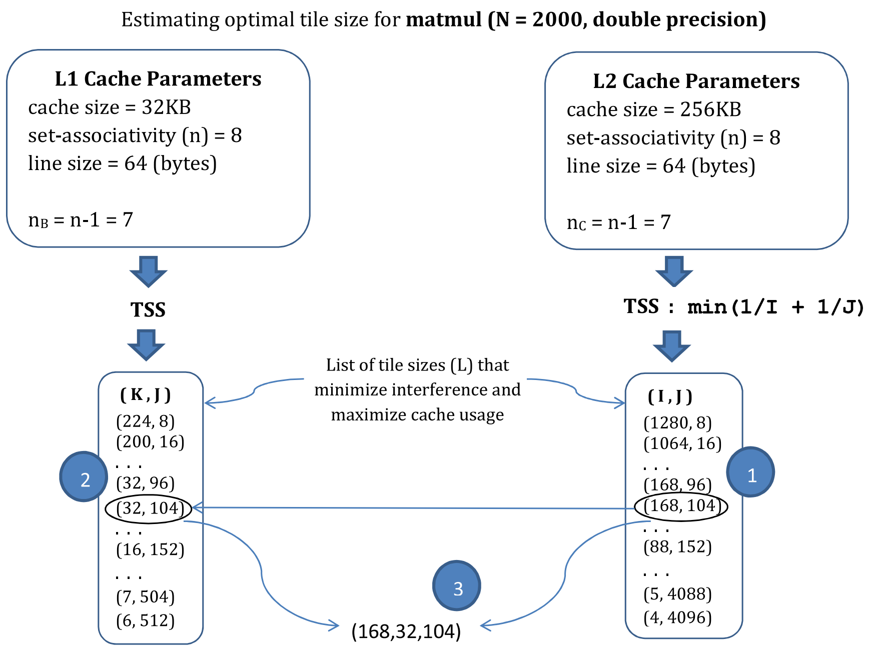


Figure 3.3: Choosing a tile size that exploits data reuse in both L1 and L2 cache

working set and occupies most ways in a set-associative cache. Thus, the TSS algorithm (described later in Section 3.4) works on the assumption that the array reference B occupies all but 1 way of the L1 cache and returns a list of (K, J) tuples all of which correspond to maximal tiles, i.e. tiles whose sizes can be increased no further without incurring conflict misses. The best (K, J) tuple and the size of the I dimension are chosen by considering further criteria.

We concluded from Equation 3.9 that tile dimensions I and J should both be large for effective data reuse in the L2 cache. Further, we observe from Figure 3.3 that the product $J * K$ is nearly constant in each of the (K, J) tuples that minimize conflict misses in the L1 cache. Thus, for J to be large, a correspondingly smaller K must be chosen. With a large J (and a large I) and a small K, it can be concluded that the RR for L2 cache as given by Equation 3.11 will also tend towards 1. This shows that the working set that should fit L2 cache is also dominated by a single array reference, C, that has reuse in the innermost inter-tile loop, kT. In such a scenario, the TSS algorithm assumes that the array reference C occupies all but 1 ways of the available ways in the L2 cache, and similarly yields a list of (I, J) tuples all of which correspond to maximal tiles in L2 cache. From the list, the tuple (I, J) that minimizes $(\frac{1}{I} + \frac{1}{J})$ is selected in accord with Equation 3.9 (marked as step **1** in Figure 3.3). For the chosen J, a corresponding K is chosen from the list of tuples (K, J) obtained earlier (marked as step **2** in the figure). Thus, the 3 tile dimensions (I, J, K) are determined using the TSS algorithm that considers reuse at both the L1 and L2 cache (marked as step **3** in the figure). Although we have only considered reuse in the L1 and L2 caches in this work, the latest microarchitectures employ a shared L3 cache and a similar analysis can be extended to analyze reuse in the L3 cache.

3.3.4 Interaction with vectorization.

Short-vector SIMD instruction sets such as AltiVec and SSE (and now AVX) have proved to be promising for enhancing performance on modern processors. It is thus critical to consider the interaction of tiling with the SIMD unit on the host architecture.

As a result of loop tiling, all loops including the vector loop are strip-mined, i.e. they are fragmented into smaller segments or strips. While this helps data reuse, it reduces the length of the vector pipeline. Thus, strip-mining the vector loop deprives the vector unit of the needed fodder for *Instruction Level Parallelism* (ILP). This effect is more pronounced in the latest microarchitectures as they have larger vector registers requiring longer loops and multiple vector load/store units that reveal more opportunity for ILP.

To make the best use of a bad bargain, the tile size should be so chosen that while data reuse is exploited, losses from a reduced vector pipeline length are minimized. This is ensured when the tile dimension corresponding to the vector loop is large, but only large enough that the tile doesn't cause interference misses. We evaluate this interplay between vectorization and data reuse in 3 different scenarios. The 3 scenarios are shown in Figure 3.4, where loops j, i and k are the vector loops in *matmul*, *trisolv* and *strsm* benchmarks, respectively. In all the 3 benchmarks, the *inter-tile* loops are in the same order (iT-jT-kT), but the order of *intra-tile* loops is different in each case and is decided by the PLuTo compiler such that efficient vectorization could be achieved. The PLuTo compiler applies the loop interchange transformation within the polyhedral framework to achieve this and as a result, any of the 3 loops could be made the innermost (vector) loop. We thus consider all the 3 possible scenarios.

According to Equation 3.11, tile dimensions I and J should be large for effective data reuse in the L2 cache. This further implies that the tile dimension K should be small for effective data reuse in the L1 cache. Thus, in *matmul* and *trisolv* benchmarks, where loops j and i are the vector loops respectively, the goals of achieving efficient vectorization and data reuse are concomitant. However, this is not the case for the *strsm* benchmark, where loop k is the vector

```
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    for(k = 0;  k< N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Matmul benchmark

```
for (iT=0; iT< N/I; iT++)
 for (jT=0; jT< N/J; jT++)
  for (kT=0; kT< N/K; kT++)
   for  (i=I*iT; i<I*iT+I-1;i++)
    for  (k=K*kT ;k<K*kT+K-1;k++)
     /* vector loop */
      for  (j=J*jT ;j<J*jT+J-1;j++)
       C[i][j] += A[i][k] * B[k][j];
```

Tiled matmul

(a)

```
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    for(k = 0;  k< j; k++)
      B[j][i] = B[j][i] – L[j][k]*B[k][i];
```

Trisolv benchmark

```
for (iT=0; iT< N/I; iT++)
 for (jT=0; jT< N/J; jT++)
  for (kT=0; kT< N/K; kT++)  {
   . . . .
   for  (j=J*jT ;j<J*jT+J-1;j++)
    for  (k=K*kT ;k<K*kT+K-1;k++)
     /* vector loop */
      for  (i=I*iT; i<I*iT+I-1;i++)
       B[j][i] = B[j][i] – L[j][k]*B[k][i];  }
```

Tiled trisolv

(b)

```
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    for(k = i+1;  k< N; k++)
      if(k==i+1) B[j][i]/=A[i][i];
      B[j][k] -= A[i][k]*B[j][i];
```

Strsm benchmark

```
for (iT=0; iT< N/I; iT++)
 for (jT=0; jT< N/J; jT++)
  for (kT=0; kT< N/K; kT++)  {
   . . . .
   for  (i=I*iT; i<I*iT+I-1;i++)
    for  (j=J*jT ;j<J*jT+J-1;j++)
     /* vector loop */
      for  (k=K*kT ;k<K*kT+K-1;k++)
       B[i][k] -= A[j][k]*B[i][j];  }
```

Tiled strsm

(c)

Figure 3.4: Vectorization, reuse and tile size

loop. Choosing a small value of K would severely hurt vectorization. However, choosing a large value of K although prevents data reuse in the L2 cache, data reuse within the L1 cache can still be achieved in addition to effective vectorization. Empirical verification supports that in such cases, 2D tiling that aims at data reuse in only L1 cache and effective vectorization, gives the best performance. This is incorporated in our framework by choosing the largest *rectangular* tile from the list of tile size tuples returned by our algorithm for the L1 cache; the outermost tile dimension is effectively left untiled by choosing its size to be the same as the problem size.

Also, since the TSS algorithm chooses each tile dimension to be a multiple of the cache line size (for achieving spatial locality), the tile dimension corresponding to the vector loop is ensured to be a multiple of the vector size. This leads to effective vectorization.

3.3.5 Interaction with Time Skewing.

For stencil codes such as *jacobi-2d* (shown earlier in Figure 3.1d), *time skewing* can significantly improve temporal locality. For time-skewed codes, however, estimating the optimal tile size is more involved because of the *shifting tiles* as shown in Figure 3.5 - the tile space shifts by the corresponding skewing factors (S_H and S_W in case of 2-D tiles) in each dimension for each iteration of the tiled time loop (loop t in Figure 3.1e). This leads to the following essential differences in the case of time-skewed codes:

1. In non-time-skewed codes, adjacent tiles access distinct data (except for some possible

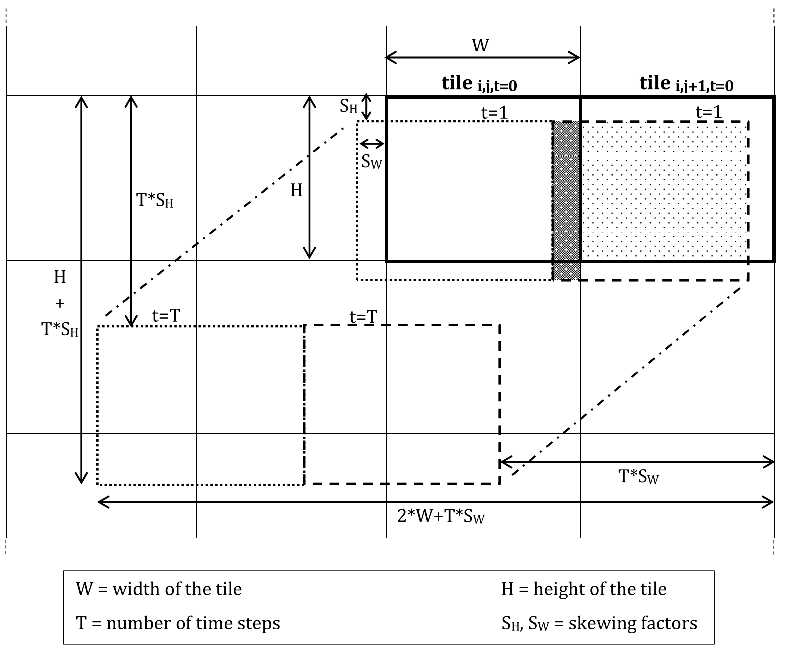


Figure 3.5: Illustration of skewed-tile traversal

overlap at tile boundary). However, as a result of *shifting* tiles in time-skewed codes, there is an opportunity for reusing common data between adjacent tiles. For example, $tile_{i,j}$ and $tile_{i,j+1}$ access common data for $t = 1$, shown by the darkly shaded region in Figure 3.5. Thus, if the data accessed by $tile_{i,j}$ in all T (length of the tiled time loop, t) time steps fits the cache, the common data can be reused by $tile_{i,j+1}$ in all T iterations. In such a scenario, the only misses incurred by $tile_{i,j+1}$ for $t = 1$ (or for $t = k$, in general) are due to the new data accessed as shown by the unshaded region in $tile_{i,j+1,t=1}$ in the figure. Thus, the number of misses are given by $\frac{W*S_H}{CLS}$, where CLS is the Cache Line Size in terms of the number of elements. The total misses incurred in time steps 1 through T are given as $\frac{W*S_H*T}{CLS}$. Similarly, the total misses incurred in the program that has a total of $\frac{N}{W} * \frac{N}{H}$ tiles and that iterates for T_0 time steps, is given as

$$\frac{S_H * T_0 * N^2}{CLS * H} \tag{3.12}$$

Since the total cache misses are inversely related to the tile height, the *TSS* algorithm chooses a tile with largest height such that the tile width is equal to a single cache line, from the list of tile size tuples that minimize interference in L1 cache. This ensures that $tile_{i,j}$ for any $t(= k)$ fits the L1 cache. However, to enable inter-tile data reuse in L2 cache, the data accessed by $tile_{i,j}$ in time steps 1 through T should fit the L2 cache. In addition, more data is brought into the cache by $tile_{i,j+1}$, with the possibility of interference in the L2 cache. As can be seen from the figure, any two consecutive tiles such as $tile_{i,j}$ and $tile_{i,j+1}$ access data within a block of dimensions (2*W+T*S$_W$)x(H+T*S$_H$), for all T iterations of the tiled time loop. Thus, the TSS algorithm assumes the worst case scenario, and chooses T such that this block fits in the L2 cache. This also minimizes interference in the L2 cache. With W and H known, the value of the tile in the time dimension, T, is thus calculated, and the tile chosen achieves reuse both in L1 and L2 caches.

2. The discussion above favors tiles with smaller widths to achieve data reuse in the L2 cache for stencil codes. This, however, is unproductive for efficient vectorization as explained. Thus, if a stencil code is vectorizable in the innermost loop, the two factors conflict with each other. Empirical results show that for stencil codes that are not vectorizable because of dependences such as *seidel*, smaller tile widths perform better, and vice-versa for vectorizable stencils such as *jacobi* and *fdtd*. Thus, for vectorizable stencils, we choose tiles with larger widths (such that the tile height is no less than a cache line) in favor of effective vectorization. In such cases, since inter-tile data reuse in the L2 cache is not achieved, the height and width of the tile are chosen such that a single tile fits L2 cache and the time dimension is chosen to be the total number of time steps.

3.3.6 Other factors that influence tiling.

Interaction with multi-level TLB. The authors in (29) argue that in addition to data reuse in the L1 cache, the tile size chosen must also minimize TLB misses. Their rationale was based on the assumption that TLB misses are much more costly than cache misses (and also possibly quite frequent given a small TLB). However, the recent architectures employ a 2-level TLB. For example, Intel's Sandy Bridge has a 64-entry L1 TLB and a 512 entry L2 TLB, while there was just a single level 64-entry TLB in the Intel's Netburst microarchitecture. On Sandy Bridge that uses a 4-level page table, the cost of a page walk is also reduced to merely 30 cycles, which is same as the latency for an L2 cache miss. In addition, tiling compilers such as PLuTo (used for this work) permute the intra-tile loops such that most references achieve spatial locality and effective vectorization in the innermost loop. This also minimizes the TLB misses as for such references, consecutive elements reside on the same page. It is for these reasons that empirical results reveal minimal impact of TLB misses on tile size selection.

Interaction with shared caches. *Chip Multiprocessing* (CMP) and *Simultaneous Multithreading* (SMT) are the two techniques currently employed to extract the inherent Instruction-level parallelism in programs run in a *multithreaded* environment. In a multithreaded environment, multiple cores may bring in different data to the shared cache on a chip multiprocessor, and similarly, multiple threads may bring in different data to the private cache on a core using the SMT technology. We account for this in our algorithm by adjusting the set-associativity to $\left(\frac{1}{T}\right)$ of the actual value, where T is the number of threads/cores that share the cache. This is done to prevent any cache conflicts even for the worst case when all T threads/cores execute different tiles and thus bring different data into the cache.

Data re-layout of the tiles/To copy or to not copy. *Copy optimization* or *copying* is a technique to adjust the data layout for reducing cache conflicts within a tile, wherein a tile is copied into a temporary *linear* array and copied back to its original memory location after execution. As a result, all elements within a tile are mapped to contiguous locations, instead of disparate locations in the cache. This data re-layout improves the cache behavior by minimizing conflict misses (33). If we thus re-layout the tiled arrays, the tile size is then largely governed by cache capacity and is easily estimated (30). However, production quality compilers such as GCC and ICC do not support such data re-layout because (1) the additional cost of copying may offset the savings, (2) there is a need to write complex clean-up code when the problem size is not a multiple of the tile size, and (3) the complexity of the transformation itself, i.e. it must be determined what to copy and when to copy. Also, another important reason is that copying cannot be performed for stencil codes, as the tiles shift in every iteration of the tiled time loop.

The TSS algorithm avoids copying. It instead achieves the purpose of minimizing interference by analyzing the source program and its interaction with the architecture and the compiler. It thus saves the overhead of copying and relieves the compiler from the burden of performing the complex copy optimization.

Problem size and array padding. As discussed earlier in Chapter 2, conflict misses are pronounced for problem sizes that are a power of 2. Such problem sizes are called *pathological* problem sizes as the various algorithms employed for tile size selection yield either too skinny or too fat tile sizes, that result in suboptimal performance. Array padding is a technique proposed in previous works (27; 28) to handle such cases, where the problem size (particularly, the leading dimension) of the arrays is increased to prevent conflicts in the cache, and achieve reasonable performance even for the pathological problem sizes. As for copying, compilers such as GCC and ICC do not support padding because (1) it transforms the array structure that must be reflected in the entire code, (2) it cannot be directly incorporated into a library routine as the problem size is not known apriori; copying the arrays into another array with padding helps, but copying has its own disadvantages and overheads.

3.4 The Tile Size Selection (TSS) Algorithm

As described in Sections 3.3.1 and 3.3.2, only references such as the array reference B for the L1 cache (and reference C for the L2 cache) in the example of *matmul*, access reusable data and dominate the working set in the respective caches. For a given source program and cache parameters, the TSS algorithm emulates cache behavior for one such array reference (say, k) during tile execution. For the emulation, the reference k is allowed to occupy $n-1$ ways of an n-way set-associative cache, while the remaining references that have small contributions to the working set are assumed to occupy the remaining 1 way of the cache. If there are K such array references, cross-interference among them is prevented by assuming the effective set-associativity of the cache to be $n_e = \lfloor \frac{n}{K} - 1 \rfloor$ instead of the actual value of n. This emulation is done for different values of the tile width[3] that are multiples of the cache line size, and the algorithm returns in each case, a corresponding tile height at which cache conflicts ensue. Thus, a list, L, of tile sizes that aim to minimize interference and maximize cache usage is generated. The same algorithm is used to determine the list of tile size tuples (K, J) and (I, J) for minimizing conflicts in the L1 and L2 cache, respectively. The estimate of the best tile size (I, J, K) is determined by further criteria as described earlier in Section 3.3.3.

The algorithm emulates the fetching of cache lines into the cache for the array reference k. The emulation is done using $cache\_emu$, an array whose every element stores the count of the number of occupied ways in a set in the cache. The elements of $cache\_emu$ are initialized to a count of 0, indicating that no way of the set is occupied. The count (of a set) is increased to emulate the fetching of a cache line into the set. A row in the tile maps to a set given by $nset$

[3]Here, the algorithm is presented for the case of 2D tiles for understandability, and is extended for the general case of nD tiles in Section 3.4.1. Width in general represents the size of the tile in leading dimension.

ALGORITHM 1: Tile Size Selection

```
**INPUT:**
Size of cache: $cSize$
Set-associativity of cache: $n$
Size of cache line: $lSize$
Leading dimension, or Problem-size: $N$
No. of arrays with self-temporal reuse in outermost loop: $K$
No. of active threads per core: $T$
No. of elements in a cache line: $CLS \left(= \frac{lSize}{size(DataType)}\right)$
No. of sets in cache: $nSets \left(= \frac{cSize}{lSize*n}\right)$
Effective set-associativity: $n_e \left(= \lfloor \frac{n}{K*T} - 1 \rfloor\right)$
**Step 1:**
**for** $w = 1$ to $min(\frac{cSize}{lSize}, \frac{N}{lSize})$ **do** {/* $w$ is tile width in number of cache lines */}
  $r = 0$
  **repeat** {/* iterates over rows in a tile */}
    $nset \leftarrow \left(\frac{r*N}{CLS}\right)\%nSets$
    **for** $c = 0$ to $w$ **do** {/* brings a tile-row into cache */}
      **if** $cache\_emu[nset + c]$ = $n_e$ **then**
        $h = r$ {/* tile height is maximum rows in tile */}
        **break** {/* interference begins at this point */}
      **else**
        $cache\_emu[nset + c] + +$
    $r + +$
  **until** $r = N$
  **if** $h < CLS$ **then**
    Add the tile size $(h, w * CLS)$ to list $L$
  **else**
    Add the tile size $(\lfloor \frac{h}{CLS} \rfloor * CLS, w * CLS)$ to list $L$
**OUTPUT: List of tile size estimates,** $L$
```

as shown in the algorithm. Since the memory addresses wrap-around the cache, our algorithm can safely assume that the first element of an array maps to the first line (set) of the cache. Henceforth, for a chosen width of w cache lines in a tile-row, the algorithm brings the lines comprising a tile-row into the cache. The cache is thus filled row-wise until all the available ways (given by n_e) in a set of the cache are filled, and interference begins to replace reusable data in the cache. Thus, the tile height (h=r, the number of rows in tile at this point) for the chosen tile width is obtained such that there are no interference misses. The outermost **for** loop over w thus generates a list of tile sizes, L. Each tile size is the tuple $(h, w * CLS)$ for a chosen value of w. The tile height in each tuple is truncated to the nearest multiple of the cache line size for efficient utilization of the spatial reuse.

3.4.1 Scope of the algorithm

This section describes the nature of programs that can benefit from the TSS algorithm. Firstly, it is important to note that programs containing references that carry reuse in one of the loops in the loop nest, can alone benefit significantly from the loop tiling transformation, and from a good tile size chosen by the TSS algorithm.

The TSS algorithm relies on exploiting reuse at two loop levels - the outermost intra-tile loop and the innermost inter-tile loop. Thus, for a program to benefit from TSS, it must contain an array reference that has reuse in the outermost intra-tile loop and one that has reuse in the innermost inter-tile loop. The TSS algorithm gives a list of good tile dimensions corresponding to the inner loops based on considering intra-tile reuse but the size of the outermost tile dimension must be obtained after considering the inter-tile reuse. The inter-tile reuse, however, depends significantly on the loop order - if the innermost inter-tile and innermost intra-tile loops correspond to the same loop in the original program as in the *strsm* kernel, inter-tile reuse is sacrificed in favor of intra-tile reuse and effective vectorization. In such a case, the outermost intra-tile loop is left untiled. In all other cases, effective inter-tile reuse can be achieved without hurting vectorization by using the cost function obtained from Equation 3.9. This strategy proves applicable for kernels that yield non-skewed rectangular tiles, such as those in basic linear algebra (BLAS), data mining and image processing applications.

In stencils, the inner space loops are skewed with respect to the outermost time loop. Thus, while intra-tile reuse can be achieved in a manner similar to the non-skewed tiled codes, inter-tile reuse is achieved in a different way, as explained in Section 3.3.5.

TSS algorithm for imperfectly nested loop nests. An important point to note is that when imperfectly nested loops are tiled, statements are distributed into different nests at one of the outer inter-tile loops. As a result, the cost functions derived in this work for the intra-tile loops as well as the innermost intra-tile loop can still be applied to such codes. However, in such cases, multiple imperfectly nested loop nests will have to use the same tile size, which may not be good for all nests.

TSS algorithm for programs with many array references. The TSS algorithm depends on the effective set-associativity, computed as $n_e = \lfloor \frac{n}{K*T} - 1 \rfloor$. Thus, if the number of array references in the source program (K) is large (or, the number of threads sharing the cache, T, is large), all array references cannot be assigned a unique way, given the limited number of ways in the cache. In such a case, we improvise the TSS algorithm by doubling the set-associativity (n) and halving the number of available sets. While this does not guarantee conflict minimization due to the overlapping of data accessed by multiple references in the cache, it nonetheless serves as a good approximation for 2 reasons, (1) with the cache size remaining constant, it still partially emulates cache conflicts and does not fill the cache to capacity, and (2) since the number of sets remain a power of 2, it still accounts for the case of pronounced cache misses for certain pathological problem sizes.

TSS algorithm for nD tiles and non-square problem sizes. The TCM/RR analysis and the TSS algorithm presented thus far assume 3D loop-nests and 2D tiles. Thus, we refer to tile dimensions, I, J and K, and to the "height" and "width" of a tile. However, in the general case of nD tiles, the tile size corresponding to the outermost intra-tile loop (I) represents the product of multiple outer intra-tile loops. Thus, the same analysis gives information about the relative impact of each tile dimension on data reuse, for an nD tile. Similarly, the TSS algorithm is easily extended for nD tiles - tile width corresponds to the tile size in the leading dimension and tile height is split into multiple dimensions and the mapping of a cache line to a particular set in the cache is accordingly computed. Thus, a list of tile size tuples, (t_1, ... , t_n), is obtained, and one of them is chosen as an estimate of the optimal tile size based on criteria determined from the TCM/RR analysis, exactly as in the case of 2D tiles. This is shown in Section 3.6 through the example of a tensor contraction kernel, *doitgen*, that involves 3D tiles and is tiled in 4 dimensions.

Also, the TSS algorithm only depends on the leading dimension of the array reference with self-temporal reuse in the innermost loop. It is thus not restricted to *square (cubic)* problem sizes but will find an estimate of optimal tile size for any problem size.

3.4.2 The Framework

The TSS algorithm is implemented within PLuTo, a source-to-source transformation system based on the polyhedral model. The complete framework of our implementation is shown in Figure 3.6.

The PLuTo transformation framework takes as input the original source code and generates *statement-wise affine transformations*. These statement-wise transformations are representative of a composition of loop transformations including *loop tiling*. However, in the absence of a user-specified tile size, PLuTo uses a default cubic tile size of 32 to generate the updated (statement-wise) transformations through its *Polyhedral Tile Specifier*. Our algorithm, instead, interacts with PLuTo at this step to provide it with its estimate of the optimal tile size. The updated transformations with the tile size information are input to CLooG (34) to generate transformed source code, which can then be compiled on the target machine using any back-end compiler such as GCC or ICC.

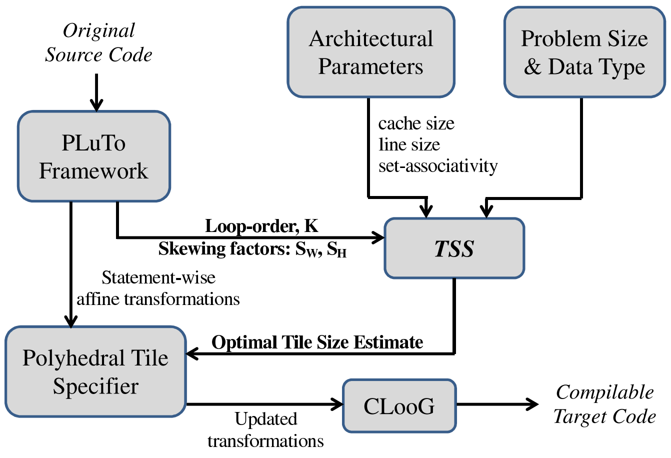


Figure 3.6: Our Framework

The TSS algorithm needs 3 architectural parameters (cache size, line size and set-associativity) and 3 program specific parameters (the order of intra-tile loops, K i.e. the number of arrays with self-temporal reuse in a particular loop, and the skewing factors). The cache parameters can be determined through the OS, e.g. by a kernel call to '*GetLogicalProcessorInformation*' function in Windows or through specific files in '*/sys/devices/system/cpu/cpu0/cache/*' in Linux. They can also be obtained using micro-benchmarks such as those in ATLAS. In our implementation, we used the former option to determine the cache parameters. The information about the program specific parameters is obtained from the PLuTo transformation framework. The order of the intra-tile loops is particularly needed, as PLuTo might interchange the loops to favor vectorization and this information is needed to determine the size of the tile in each dimension, as explained in Section 3.3.6. PLuTo also provides information about the skewing factors and loop vectorization for codes that are time skewed, to determine the tile size for such codes as discussed in Section 3.3.5.

3.5 Experimental Setup

Microarchitecture	Cache Size (L1|L2|L3)	Cache Type (L1|L2|L3)	Set-associativity (L1|L2|L3)
Sandy Bridge	(32KB|256KB|12.5MB)	(priv.|priv.|shared)	(8|8|20)
Core	(32KB|1.25MB|-)	(priv.|shared|-)	(8|8|-)

Table 3.1: Details of the microarchitectures

We tested the effectiveness of our algorithm on an Intel Xeon E5-2650 processor with 8 cores (running at 2 GHz) based on the Sandy Bridge microarchitecture, and an Intel Core2 E6400 processor with 2 cores (running at 2.13 GHz) based on the Core microarchitecture. The details of each microarchitecture is given in Table 3.1[4]. For our experiments, we used 12 kernel benchmark programs from PLuTo and PolyBench (36) that are listed in Table 3.2.

Kernel	Description	Problem Size	Data Type
matmul	Matrix Multiplication	N1=512; N2=2000	Double
dsyrk	Symmetric rank-k operations	N1=512; N2=2000	Double
dsyr2k	Symmetric rank-2k operations	N1=512; N2=2000	Double
lu	Lower Upper Decomposition	N1=512; N2=2000	Double
trisolv	Multiple Triangular Solver	N1=512; N2=2000	Double
doitgen	Multiresolution analysis kernel (MADNESS)	N1=128; N2=150	Double
strsm	Linear Equation Solver	N1=512; N2=2000	Float
tmm	Triangular Matrix Product	N1=512; N2=2000	Float
corcol	Correlation Computation	N1=512; N2=2000	Float
covcol	Covariance Computation	N1=512; N2=2000	Float
jacobi-2d	2-D Jacobi stencil computation	T=128; N1=512; N2=2000	Float
seidel-2d	2-D Gauss Seidel stencil computation	T=128; N1=512; N2=2000	Float

Table 3.2: Summary of the benchmarks

Of these 12 kernel benchmarks, the first 8 are various linear algebra kernels and solvers,

[4]In the table, we list the effective L2 cache size as presented in (35)

the next two (*corcol* and *covcol*) are kernels employed in data mining programs and the last 2 (*jacobi* and *seidel*) are stencil kernels. Since the tile size varies with the problem size, we tested the algorithm for 2 different problems sizes - one that is a power of 2, and the other that is not. The problem sizes that are a power of 2 are termed as ‘pathological’, because they cause pronounced conflict misses for certain tile sizes, whereas all other problem sizes do not demonstrate such a behavior. It is for this reason that we chose a problem size in each of these 2 categories. In addition, since the tile size also depends on the data type of the problem arrays, we show the performance results of our algorithm for different benchmarks, some have single-precision while the others have double-precision data arrays. All tiled codes used in the experiments were generated using PLuTo (version 0.9.0) using the options ‘–tile’ and ‘–parallel’ (for extracting pipelined parallelism in time-skewed stencils). For all our experiments, we used the Intel C Compiler (ICC v13.0.1 using ‘-O3’ optimization option) as the backend compiler to compile the transformed source programs. We used the ‘-parallel’ compiler option with ICC only to obtain results on the performance of the TSS algorithm on multiple cores.

3.6 Experimental Results

Among past works that used an analytical model for tile size selection, most algorithms suffered from poor performance for the ‘pathological’ problem sizes. As a result, later models such as *eucpad* (27) mentioned in Table 3.3, use padding to overcome these pronounced conflict misses that result at such problem sizes. However, *eucpad* considers neither the impact of set-associativity nor of data reuse in multiple levels of cache. The TSS algorithm, on the other hand, considers the impact of set-associativity in generating the list of maximal tile sizes for both the L1 and L2 cache. The TSS algorithm further considers data reuse in L1 and L2 cache to generate its estimate of the optimal tile size as discussed in Section 3.3. We compare *eucpad* and *TSS* in Table 3.4.

Algorithm	Cost function	Pad Size		Set-associativity	Data reuse in L2
		single prec.	double prec.		
eucpad	$min\left(\frac{1}{K}+\frac{1}{J}\right)$	6	7	Not considered	Not considered
TSS	$min\left(\frac{1}{I}+\frac{1}{J}\right)$	-	-	Considered	Considered

Table 3.3: Comparison of the algorithms

The *eucpad* algorithm chooses non-conflicting tile widths using the Euclidean GCD algorithm (3) and tile heights using a simple recurrence. For pathological problem sizes, it allows for a padding of 0-7 elements and chooses a tile size from a list of sizes obtained by considering the different padding options. The cost function employed by *eucpad* tends to choose a square tile from the list of non-conflicting tile sizes obtained. It must be noted that their cost function is precisely based on Equation 3.4, that minimizes misses at the L1 cache. However, since the impact of set-associativity is ignored, the tile size chosen does not minimize the conflict misses. For example, *eucpad* chooses a tile size of (h,w)=(40,88) for the *matmul* kernel. The TSS algorithm, on the other hand, generates (40,80) as one of the maximal tiles for the L1 cache. In other words, the TSS algorithm predicts pronounced interference misses at the tile size (40,88), which is actually observed - the L1 cache misses incurred by the tile size (40,88) are 1.4 times more than those incurred by the size (40,80).

Table 3.4 also compares the tile size chosen by the TSS algorithm with the "Best Tile". To generate the *Best Tile*, we extensively ran all combinations of tiled codes with each tile dimension that is a multiple of cache line size, ranging from a cache line size (8 for double precision and 16 for single precision floating data) to $\sqrt{\frac{L2CacheSize}{size(DataType)}}$ (176 for double precision and 256 for single precision floating data). The reason for choosing this range of tile sizes is that the working set that results from these sizes is large enough to occupy the L2 cache, which is essential to compare with the TSS algorithm. The search space is chosen considering the L2 cache size on Sandy Bridge, as a large shared cache in Core microarchitecture would yield an unmanageable search space (the search space with the chosen limits already leads to 10648 tiles for double precision and 4096 tiles for single precision floating point data arrays). However, still, the *Best Tile* obtained for both the microarchitectures clearly demonstrates the various factors impacting the tile size, and that the tile size generated by the TSS algorithm is close-to-the-optimal tile size.

Table 3.4 shows that for linear algebra and data mining kernels with 3D loop-nests, the tile size chosen by TSS, like that of *Best Tile*, are rectangular and are such that the outermost and innermost intra-tile dimensions (I and J respectively) are large. This favors both vectorization and data reuse in the L2 cache as discussed in Section 3.3. Further, since the chosen tile size does not incur any interference misses, the performance achieved by TSS is always close to that achieved by the *Best Tile*. Since the tile sizes chosen are rectangular, in some cases the optimal tile size cannot be captured within the already large search space and thus, the tile size chosen by TSS could even outperform the *Best Tile*. This is particularly observed for the Core microarchitecture that has a considerably large L2 cache. From the table, an important observation can be made about the *Best Tile* sizes - the tile dimension corresponding to the innermost (vector) loop (96, for example, in *matmul*) is not the maximum possible length (176, for *matmul*) as would be expected to favor vectorization. The *Best Tile*, instead conforms to the cost model used by TSS (i.e. $min(\frac{1}{I} + \frac{1}{J})$), thus validating the importance of reusing the data in the L2 cache.

For the linear algebra and data mining kernels considered, PLuTo generates tiled code in which the goals of achieving data reuse in the L2 cache and efficient vectorization are concomitant. However, as discussed in Section 3.3.6, these goals conflict in the case of tiled *strsm* benchmark, where the algorithm decides in favor of efficient vectorization and data reuse in only the L1 cache. In this case, the outermost loop is effectively left untiled by choosing the tile size for that loop to be the problem size as shown in table 3.4. Another important observation from table 3.4 is that while TSS performs significantly better than *eucpad* for most benchmarks, their performance is comparable for *lu* and *trisolv* benchmarks. The reason for this is that both of these benchmarks have non-rectangular iteration spaces, and thus the reuse distance in different tiles varies as a function of their position in the iteration space. Our algorithm underestimates the tile size in such cases. However, since no conflict misses result from underestimating the tile size, our algorithm still achieves good performance in such cases.

Among the linear algebra kernels, *doitgen* has 3D arrays and a loop-nest that is tiled in 4 dimensions. To estimate the tile size for *doitgen*, the framework first identifies the array references carrying reuse in L2 cache. The TSS algorithm (modified for 3D arrays, as discussed in Section 3.4.1) generates a list of tile size triplets, all of which represent maximal tiles that do not incur interference. From this list, the triplet that maximizes reuse in the L2 cache according to the TCM/RR analysis is chosen as an estimate of the optimal tile size. The TCM/RR analysis suggests that for a tile size triplet, (P, Q, R), the tile dimension corresponding to the innermost loop (R in this case) and the tile dimension corresponding to the outermost loop (the product $P * Q$ in this case) should both be large. Thus, the tile size triplet that satisfies these criteria is selected from the list. With P, Q and R known, the remaining tile dimension, S, is computed by considering reuse in the L1 cache, as demonstrated earlier for the case of 3D loop-nests through Figure 3.3.

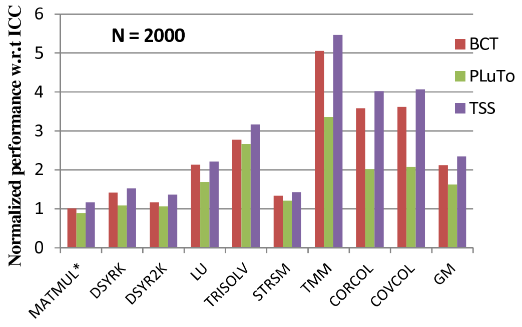
		Sandy Bridge				Core			
Benchmark	Scheme	Problem Size = 2000		Problem Size = 512		Problem Size = 2000		Problem Size = 512	
		Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)
	eucpad	40x88	3.20	64x48	3.07	40x88	2.81	64x48	1.51
matmul	TSS	168x32x104	4.1	40x7x504	3.45	248x8x328	3.2	192x7x504	1.46
	Best Tile	176x32x96	4.1	144x48x176	3.42	128x16x128	3.22	144x64x16	1.78
	eucpad	40x88	3.22	64x48	2.55	40x88	2.80	64x48	1.96
dsyrk	TSS	168x32x104	3.72	40x7x504	3.35	248x8x328	3.19	192x7x504	2.46
	Best Tile	160x16x144	3.89	96x32x176	3.22	160x144x16	3.09	128x16x176	2.31
	eucpad	40x88	3.98	64x48	2.93	40x88	2.76	64x48	2.01
dsyr2k	TSS	120x8x88	5.11	16x4x504	3.96	248x8x136	3.72	96x4x504	2.16
	Best Tile	56x8x112	5.27	176x16x176	3.90	96x16x96	3.58	176x32x176	2.20
	eucpad	40x88	2.82	64x48	2.07	40x88	1.89	48x64	1.63
lu	TSS	168x32x104	2.87	40x7x504	2.79	248x8x328	2.30	192x7x504	2.01
	Best Tile	176x16x160	3.01	96x32x176	2.63	176x16x144	2.01	160x16x176	1.86
	eucpad	40x88	3.76	64x48	2.43	40x88	2.52	64x48	2.19
trisolv	TSS	168x32x104	3.74	40x7x504	3.43	248x8x328	2.66	192x7x504	2.48
	Best Tile	176x80x128	3.81	80x32x176	2.84	176x112x176	1.80	176x176x176	2.09
	eucpad	80x64	1.52	80x64	1.49	80x64	1.18	80x64	1.17
strsm	TSS_{3D*}	2000x16x320	1.68	512x14x496	4.63	2000x16x320	1.26	512x14x496	1.96
	Best Tile	160x144x256	1.47	208x48x256	2.58	144x240x224	1.11	128x48x256	1.48
	eucpad	80x64	5.74	80x64	4.27	80x64	4.49	80x64	3.72
tmm	TSS	160x16x208	7.3	80x14x496	4.55	632x16x320	6.00	384x14x496	5.25
	Best Tile	192x16x256	7.62	160x16x256	4.02	256x16x256	5.28	48x16x256	4.70
	eucpad	80x64	4.31	80x64	3.58	80x64	2.14	80x64	2.09
corcol	TSS	160x16x208	6.16	80x14x496	4.68	632x16x320	2.69	384x14x496	2.46
	Best Tile	144x16x256	6.41	112x16x176	4.51	176x16x256	2.38	208x16x176	2.37
	eucpad	80x64	4.35	80x64	3.69	80x64	2.18	80x64	2.17
covcol	TSS	160x16x208	6.26	80x14x496	4.98	632x16x320	2.72	384x14x496	2.58
	Best Tile	112x16x256	6.51	96x16x176	4.77	224x16x256	2.41	112x16x176	2.47
	eucpad	-	-	-	-	-	-	-	-
doitgen	TSS	8x14x16x150	3.26	5x14x24x128	3.21	4x68x16x150	2.49	5x127x24x128	2.66
	Best Tile	32x8x8x104	2.92	16x8x24x120	3.07	32x8x32x80	2.38	32x24x24x120	2.52
	eucpad	-	-	-	-	-	-	-	-
jacobi-2d	TSS	128x32x560	4.82	128x48x496	3.83	128x48x2000	2.30	128x192x496	1.82
	Best Tile	64x64x32	4.73	64x32x240	3.93	128x96x256	2.45	128x224x256	1.30
	eucpad	-	-	-	-	-	-	-	-
seidel-2d	TSS	128x208x16	1.31	96x14x16	1.25	128x208x16	1.07	128x14x16	1.34
	Best Tile	64x192x16	1.32	32x64x16	1.28	32x64x16	1.15	64x208x16	1.3

Table 3.4: Performance of TSS algorithm

Of the two stencil benchmarks considered, *seidel* is representative of non-vectorizable stencils and *jacobi* is representative of vectorizable stencils. For *seidel*, it can be observed that the analysis for data reuse in the L2 cache presented in Section 3.3.6 holds true - the *Best Tile* is rectangular with a larger tile height. The tile size corresponding to the outermost time loop is determined from the formula derived earlier in the same section, and the tile size determined by TSS performs close to the *Best Tile*. For *jacobi*, however, the TSS algorithm chooses a tile size to benefit from vectorization and intra-tile reuse in the L2 cache, at the cost of inter-tile reuse in L2 cache. Results show that the TSS algorithm achieves performance comparable to the *Best Tile* in such cases as well. The difference in the optimal tile sizes for these two classes of stencils clearly validate both the impact of vectorization and reuse in the L2 cache, on tile size selection.

Comparison with other approaches

Existing compilers - Figure 3.7 plots normalized performance improvements achieved by the TSS algorithm, the default tiling in PLuTo and the *best cubic tile* (BCT) with respect to the Intel's C compiler. We show results for 9 of the 12 benchmarks considered on Sandy Bridge. The best cubic tiles have been chosen for comparison as some auto-tuning frameworks (20; 25) only consider cubic tiles to decide the best tiled code. Results show that the TSS algorithm outperforms the best cubic tile in all cases, achieving an average improvement of 9.7% and 20.4% for the 2 problem sizes. It is interesting to note that ICC could only tile the *matmul* benchmark (marked with an asterik) among these 9 benchmarks, where it chose a cubic tile of size 128. Choosing such a tile size leads to a working set size that is mid-way between the L1 and the L2 cache size. The rationale is that some data could be reused in L1 cache and if there are some misses, they could be serviced by the L2 cache. Although an intuitive approach, it cannot match the performance of the TSS algorithm since it only considers 1 level of data reuse, whereas as shown by our analysis, there are atleast 2 (loop) levels at which data can be reused (loops kT and i for the *matmul* benchmark). In case of PLuto, default cubic tiles of size 32 usually under-utilize the L1 cache and thus perform sub-optimally.



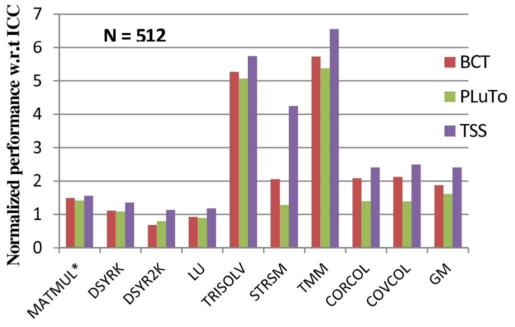


Figure 3.7: Comparison of TSS with other approaches that use 3D-tiling

The DL/ML model - The authors in (18) have recently proposed a useful analytical model for bounding the search space of tile sizes. The authors compute the *Minimum working set Lines* (ML) and the *Distinct Lines* (DL) (32) as a function of the tile size that provide the upper and the lower bounds on the tile size to ensure intra-tile data reuse in the L1 cache. The authors further use the DL model to enable inter-tile data reuse in a higher level of cache. However, using these 3 bounds and only considering tile sizes in multiples of the cache line size, we get a search space of 7931 tile sizes for the *matmul* benchmark on the Sandy Bridge processor used in our experiments. The size of the search space is similar for other benchmarks as well, which although much smaller than the original search space, is nonetheless significantly large. Our work accommodates cache set-associativity to narrow down the search space to a list of maximal tile sizes, which maximally utilize the caches without causing interference. As a result, all maximal tiles obtained by the TSS algorithm (for linear algebra kernels) lie within the search space obtained from DL/ML, and thus the performance achieved is similar in both approaches. The TSS algorithm further uses a cost model derived from general principles underlying data reuse in linear algebra (and data mining) kernels to choose a single tile from the list of maximal tiles that minimizes cache misses.

For stencil codes, however, inter-tile data reuse does not happen in the same manner as in linear algebra codes. For example, for *seidel-2d* stencil code, the lower bound according to DL model on the Sandy Bridge processor (with 32KB L1 cache) is given as, $(H+T+1)*(\lceil \frac{W+T}{CLS} \rceil + 1) \geq 512$, where W, H are tile width and height, respectively, and T is the tile size corresponding to the outermost time loop. Using this lower bound, the best tile size (32x64x16) for the problem size of 512 and various other close-to-the-best tile sizes including that chosen by TSS lie outside the search space of DL/ML. This is because, in the optimal tile, the tile height should be less to avoid interference misses, and the tile width should be less to gain inter-tile reuse in the L2 cache on account of *shifting* tiles, as discussed in Section 3.3.5. The DL/ML model does not consider either and thus cannot adequately bound the search space. The tile size chosen by the TSS algorithm considers these factors affecting data reuse and achieves good performance.

Copy and Tile - Table 3.5 compares the TSS algorithm with '*Copy and Tile*' (abbreviated as '*C+T*') for 3 out of 6 level-3 BLAS kernels on Sandy Bridge. 'Copy and Tile' emulates the *data copying* performed in ATLAS, where the innermost matrix is copied in a block-major layout as described in (20) to prevent conflict misses. After data copying, a cubic tile is chosen that maximizes the L1 cache usage based on the inequality derived in (30), which is a refinement over (20). Further, we also ensure spatial locality for the matrices as in ATLAS, and choose the tile size to be a multiple of the problem size to avoid complex clean-up code. We do not, however, compare with ATLAS, as ATLAS also performs other optimizations such as register tiling and pipeline scheduling in addition to tile size selection.

Results in Table 3.5 show that the TSS algorithm achieves a performance improvement of (17%) over ‘C+T’ for problem sizes that are not a power of 2 (2000 and 4000). This improvement is attributed to (1) data reuse in L2 cache exploited by the tile size chosen by TSS whereas ‘C+T’ cannot capture this opportunity and results in 4% higher L2 cache miss rate than TSS, and (2) since ‘C+T’ only considers cubic tile sizes, it cannot take advantage of the large vector pipeline length made possible by considering rectangular tiles.

Benchmark	Scheme	N=2000		N=4000		N=256		N=1024	
		Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)
matmul	TSS	168x32x104	4.1	168x32x104	4.28	40x7x504	3.73	40x7x504	3.52
	C+T	50x50x50	3.74	50x50x50	3.89	64x64x64	3.73	64x64x64	4.38
dsyrk	TSS	168x32x104	3.72	168x32x104	3.60	40x7x504	2.85	40x7x504	3.58
	C+T	50x50x50	3.02	50x50x50	2.80	64x64x64	2.72	64x64x64	3.71
dsyr2k	TSS	120x8x88	5.11	120x8x88	4.98	16x4x504	3.87	16x4x504	4.30
	C+T	40x40x40	4.14	40x40x40	3.75	32x32x32	3.30	32x32x32	2.59

Table 3.5: Comparison between TSS and ‘Copy and Tile’

On the other hand, the performance of ‘C+T’ and TSS is comparable for the pathological problem sizes (256 and 1024), with the former even outperforming TSS in the case of matmul. This is because, to avoid interference for such problem sizes, the TSS algorithm chooses ‘thin’ tiles, (for example, the tile size chosen for *matmul* at N=1024 is 28x7x504). This leads to less effective data reuse in the L2 cache, as compared to square tiles chosen by ‘C+T’. The reason that ‘C+T’ could choose square tiles even for these pathological problem sizes stems from the fact that it is immune to the effects of problem size as a result of the copy optimization. Data copying, however, is a complex optimization and is not supported by production-quality compilers such as GCC and ICC. Since the TSS algorithm does not rely on copy optimization, it can be readily implemented in any production-quality compiler.

Performance of the TSS algorithm in a multithreading environment

Table 3.6 shows the performance of TSS algorithm using 4 and 8 threads on the Intel Xeon processor based on Sandy Bridge microarchitecture. For the purpose of evaluation, we ran the same 10648 different tiled versions of *matmul* and *dsyr2k* benchmarks to obtain the “Best Tile”. In the table, we compare the performance of the TSS algorithm with the *Best Tile* and the *best cubic tiled* (BCT) code.

On Sandy Bridge that allows 2-way hyperthreading, 2 threads on a core share resources including private L1 and L2 caches on the core and bring in data simultaneously to the caches.

	matmul (N=2000)				dsyr2k (N=2000)			
	4 threads		8 threads		4 threads		8 threads	
Scheme	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)	Tile Size	Perf. (GFLOPS)
BCT	64x64x64	7.48	64x64x64	13.96	64x64x64	6.06	64x64x64	10.07
TSS	128x8x80	7.84	128x8x80	15.3	40x8x40	6.33	40x8x40	10.53
Best Tile	72x16x120	8.12	64x16x96	15.63	72x8x72	7.02	96x16x96	11.23

Table 3.6: Performance of TSS algorithm in a multi-threaded environment

This is incorporated in the TSS algorithm by reducing the effective set-associativity by a factor of 2 or the number of threads sharing the cache. As a result, a smaller tile size is chosen by our algorithm when using 4 and 8 threads (run on 2 and 4 cores, respectively), performs comparable to the *Best Tile* and outperforms the best cubic tiled code. A smaller tile size is similarly observed for the best cubic tiles using this 2-way hyperthreading technology as compared to that observed when using a single thread per core, which further validates our proposed effect of multithreading on tile size selection. As a result of using smaller tiles instead of the same estimates of tile size for a single core, the TSS algorithm achieves an improvement of 28% and 7%, respectively, for *matmul* and *dsyr2k* benchmarks. A similar phenomenon is observed (data not shown), when multiple cores share the L2 cache on the Core2 processor using the Chip Multiprocessing (CMP) technology.

3.7 Related Work

In the past, the problem of optimal tile size selection has been attempted using two approaches, (1) through an analytical framework based on analyzing the interaction between the source program and the host architecture, and (2) through an extensive search at runtime to tune for tile size. In approach (1), there has been considerable work in the past (13; 14; 3; 15; 4; 27; 29). However, none of these works has proved to be effective for a range of programs. The authors in (13; 14; 15) study the impact of problem size and cache parameters on the tile size to eliminate self-interference misses, but do not consider cross-interference misses. On the other hand, the authors in (3; 4) aim to minimize cross-interference misses also. However, they assume the cache to be direct-mapped and use probabilistic approaches for minimizing conflicts that leads to sub-optimal performance, especially for the 'pathological' problem sizes. The authors in (27; 28; 29) address this problem of robustness by using array padding for such pathological problem sizes. However, these works do not accommodate set-associative caches in their models, nor consider the option of exploiting data reuse in the multiple levels of cache. The authors in (37) use Presburger formulas to express cache misses and can consider moderate levels of associativity, but cannot accommodate the high set-associativity in modern caches.

Of all the works in approach (2), the most prominent work is the ATLAS library generator (20), which generates optimized BLAS library on a given target platform. However, ATLAS relies on an extensive time-consuming search for this purpose. Other self-tuning library generators (19; 21; 22) suffer from the same problem of large search space. Tiwari et. al. (38) combine the CHiLL (39) compiler framework with Active Harmony (40) to manage the cost of this search space, but rely on the user for the copy optimization. Yotov et. al. in (30) improve upon ATLAS by showing that search is not necessary to generate optimized BLAS. However, they use the code generator of ATLAS, which relies on the adhoc copy optimization to reduce interference and thus estimate the optimal tile size. The automatic tile size creation model of Yuki et al. (25) avoids array copying but only considers cubic tiles to contain the data collection time. But, the authors in (41) and (42) show that, for linear algebra and stencil codes, respectively, the optimal tile is not cubic, which is also corroborated by our work.

Chen et. al. (43) and Shirako et. al. (18) combine an analytical model with empirical search to manage the search space. Chun et. al. rely on copying to work with a reduced search space and use heuristics to traverse it. Shirako et. al. propose a model to determine the upper and lower bounds on the search space using the existing Distinct Lines (DL) model (16) and their Minimum Working Set Lines (ML) model, and consider data reuse in multiple levels of cache. However, the ML model assumes an optimal cache replacement policy, thus, effectively ignoring the impact of set-associativity in caches. In this work, we accommodate the impact of set-associativity in our analysis of data reuse.

Recently, the auto-tuning framework has been improved to select the optimal tile size while in the production run (ETile (23)) using parameterized tiled code (44). ETile monitors a few loop iterations for a tile size to determine the optimality. However, this doesn't reduce the search space and may not give the best tile as it is difficult to capture reuse in a few iterations, especially if data reuse is in the outermost loop.

3.8 Conclusion

This work revisits the problem of tile size selection in the context of modern processors. It proposes an analytical framework that unveils several factors that had not been considered by previous models. It particularly considers the data reuse in not just the L1, but also the L2 cache. In addition, it considers the interaction of the SIMD unit in modern processors with tile size selection. The TSS algorithm proposed chooses rectangular tiles that benefit from both vectorization and data reuse in multiple levels of cache. Experimental results indicate that the TSS algorithm achieves significant improvement over the previous analytical models that did not consider these important factors on modern processors. In addition, results indicate that the tile size chosen by the TSS algorithm is comparable to exhaustive search in a constrained search space. Since the TSS algorithm estimates the optimal tile size at compile-time, it can be readily incorporated within any production-quality compiler.

Chapter 4

Coordinated Multi-Stage Prefetching for Present-day Processors

4.1 Introduction

Data prefetching proves very effective for hiding the large memory latency. In many cases, the applicability of other memory optimizations such as loop fusion and loop blocking is limited, and thus prefetching proves particularly useful. It is for this reason that latest processors are equipped with improved hardware prefetching logic, and their instruction sets provide support for software prefetch instructions that can selectively prefetch data to different levels of cache. However, while such an extensive support exists in the form of hardware, software, or coordinated hardware-software prefetching, the choice of the right prefetching strategy among the various options continues to remain a mystery for programmers and compiler writers.

The choice of the right prefetching strategy is a function of the various features supporting prefetching in the host architecture. One such feature is the nature of hardware prefetcher - its effectiveness, aggressiveness and behavior in the presence of software prefetch instructions. For example, on SandyBridge, the Mid-level cache (MLC) streamer hardware prefetcher is very effective and prefetches data to the L2 cache. Unlike software prefetching, it is not limited by the small size of the Line Fill Buffer at the L2 cache that allows very few outstanding prefetch requests, but can effectively prefetch data for 32 streams. Further, it can be trained by the software prefetch requests to the L2 cache for prefetching data to the L1 cache. This makes for an ideal scenario for *coordination* between the MLC streamer prefetcher and the L1 software prefetches that together fetch the data all the way to the L1 cache and make up for the not-so-effective L1 hardware prefetcher. The L1 software prefetching also provides other advantages: (1) Using a larger prefetch distance helps to increase the limit of look-ahead prefetch distance of the hardware prefetcher as the existing limit of 20 cache lines proves insufficient for small loops (as in matrix multiplication). (2) The hardware prefetcher stops at page boundaries. However, using a large prefetch distance in the software prefetch instructions helps to re-trigger the hardware prefetcher in time to prevent incurring any stalls at page boundaries. (3) The hardware prefetcher can track a maximum of 32 streams. Thus, for programs with more than 32 streams, software prefetch instructions prove particularly helpful to improve the prefetch coverage by prefetching data for the remaining streams directly to the L1 cache.

On the other hand, on Xeon Phi, there is similarly an effective L2 streamer prefetcher but it does not prefetch data in the presence of L1 software prefetch instructions. Thus, the advantages of coordinated hardware-software prefetching cannot be realized on Xeon Phi. However, unlike SandyBridge that supports very few outstanding software prefetch requests at the L2 cache, each core on Xeon Phi supports a much larger number of outstanding software prefetch requests at the L2 cache. Thus, like SandyBridge, the data can be brought all the way to the L1 cache through coordinated multi-stage prefetching. But, unlike SandyBridge, the *coordination* is between software prefetch instructions at different levels of cache, i.e. data is first prefetched using a larger prefetch distance at the larger last level cache, while a smaller prefetch distance is used at the L1 cache to prefetch data from the next level cache. A smaller prefetch distance at the L1 cache allows for fewer outstanding prefetches, thus minimizing contention in the 8-entry MSHR file at the L1 cache. It is interesting to note that Xeon Phi with in-order cores and blocking caches, is more sensitive to the choice of prefetching technique than SandyBridge because pipeline stalls due to inadequate prefetching cannot be tolerated through out-of-order execution capability as in SandyBridge.

In recent past, hardware-based coordinated multi-stage prefetching has been implemented in IBM's Power6 microarchitecture (45) where the L1 hardware prefetcher coordinates with the L2 hardware prefetcher to bring the data to the L1 cache in stages. However, as noted above, software prefetches can be used in a similar manner to achieve this coordination and overcome the limitations of the existing hardware prefetchers. Also, there has been some work (46; 47; 48) that has considered employing software to aid hardware prefetching, but in such cases, the hardware was built around achieving such coordination. This, however, is not true for existing processors, and thus achieving such coordination while considering other hardware features is non-trivial. Recently, Lee et al. (49) have studied the interaction between software and hardware prefetching but have not proposed any particular prefetching strategy.

Through this work, we make the following contributions:

1. We study the influence of various hardware features on data prefetching and their impact on the choice of prefetching technique for different hardware platforms.
2. Based on our study, we propose a coordinated multi-stage prefetching algorithm for 2 different state-of-the-art processors (SandyBridge and Xeon Phi). The means of achieving the coordination, is however, different on each platform depending on the hardware supporting prefetching.
3. We evaluate the performance of the different prefetching techniques possible on both platforms and identify the reasons for their respective behaviors in different programs. Our experiments with prefetching in multithreaded environment further provides useful insights about the interaction between hardware and prefetching, and leads us to an interesting compiler optimization to tackle pronounced contention for resources in Xeon Phi. The coordinated multi-stage prefetching proposed in this work is a static compiler technique with no hardware overhead and can thus be incorporated in existing production-quality compilers, or serve as an effective tool in the hands of a programmer.

Our multi-stage prefetching algorithm works in three phases, (1) identifying *what* to prefetch, (2) identifying *where* to insert the prefetch instructions, and (3) identifying *when* to prefetch, or what prefetch distance to use at each level. The choice of prefetch instructions and the prefetch distance at each level is carefully chosen to achieve the desired coordination. We have implemented our prefetching algorithm in the ROSE source-to-source compiler framework that transforms the original source code to include the prefetch instructions. The generated transformed code is then compiled using the Intel compiler as the back-end vendor compiler. We tested our algorithm using various memory-intensive benchmarks from the SPEC CPU2006 and OMP2012 suites on Intel SandyBridge and Xeon Phi processors. Experimental results show that our multi-stage prefetching achieves a speedup of 1.55X over the Xeon Phi hardware prefetcher, and that of 1.3X over the state-of-the-art Intel compiler for Xeon Phi. On SandyBridge that employs an effective hardware prefetcher, an improvement of 1.08X is obtained.

The rest of the chapter is organized as follows. Section 4.2 reinforces our motivation through an example. We describe our multi-stage prefetching algorithm with its three phases in Section

4.3. The interaction of our prefetching algorithm with the multithreading technique is also discussed in this section. Section 4.4 describes our compiler framework for implementing the prefetching algorithm. We present experimental results and discuss the results obtained using different prefetching strategies on individual benchmarks in Section 4.5. Related work is presented in Section 4.6 and we conclude in Section 4.7.

4.2 Motivation

On all hardware platforms, the size of the LFB or MSHR file at the L1 cache is usually small. This is because any arriving data requests initiates a fully associative search across the structure to eliminate redundant requests, which is costly in terms of power usage and time delay, restricting its size. Chip area concerns also limit their size since these are located close to the core. It is for these reasons that both SandyBridge and Xeon Phi allow for only 8 outstanding prefetch requests at the L1 cache. However, clearly, such a small number of requests are insufficient to hide the large memory latencies on modern processors as pipeline stalls will result once the MSHR file or the load buffers are full. As a result, a multi-stage prefetching algorithm that brings the data from the memory in stages being cognizant of the resource availability at each level, is necessitated. This is more clearly illustrated through the following example of a well known memory-intensive weather prediction benchmark from SPEC OMP2012 suite, *swim*. The Xeon Phi processor is considered below, and similar arguments hold for a multi-core processor as well.

```
for (i=0; i<M; i++) {
 for (j=0; j<N; j++) {
   S1: UNEW[i+1][j] = UOLD[i+1][j]+C1*(Z[i+1][j+1]+Z[i+1][j])
        *(CV[i+1][j+1]+CV[i][j+1]+CV[i][j]+CV[i+1][j])-C2*(H[i+1][j]-H[i][j]);
   S2: VNEW[i][j+1] = VOLD[i][j+1]-C1*(Z[i+1][j+1]+Z[i][j+1])
        *(CU[i+1][j+1]+CU[i][j+1]+CU[i][j]+CU[i+1][j])-C3*(H[i][j+1]-H[i][j]);
   S3: PNEW[i][j] = POLD[i][j]-C2*(CU[i+1][j]-CU[i][j])
        -C3*(CV[i][j+1]-CV[i][j]);
  }
}
```

Figure 4.1: An example loop nest in the *swim* benchmark.

Figure 4.1 shows one of the three computationally (and memory) intensive loop nests in *swim*. The loop nest shown has 14 array references that access different cache lines, or in other words 14 data streams that need to be prefetched. When testing on Intel Xeon Phi that has a maximum memory latency of 1000 cycles, we observe that a minimum prefetch distance of 6 cache lines is needed to hide the memory latency. That is, there can be a maximum of 14*6 (= 84) outstanding prefetches per thread for *swim*. Thus, for the data to be prefetched directly to the L1 cache, the L1 cache should provide 84 MSHRs per thread (in the ideal scenario when there is enough off-chip memory bandwidth available), or at least much more than the currently available 8 MSHRs per thread. Such a large MSHR file at L1 cache is not feasible. To make things worse, the 8 MSHRs at L1 cache are shared by the 4 SMT threads on each core on Xeon Phi.

Thus we implement a coordinated multi-stage data prefetching strategy, where data is first brought to the lower-level cache, e.g., L2 cache (that on Xeon Phi, has more MSHRs to allow holding more requests for hiding larger memory latencies) using a large prefetch distance (6 cache lines in case of *swim*). Subsequently, data is brought from the lower-level cache (e.g., L2 cache) to the higher-level cache (e.g., L1 cache) using a smaller prefetch distance (1 cache line in case of *swim*) since the data is already in the next-level cache. As a result, a small MSHR file at the L1 cache proves sufficient to hide the small L2-to-L1 latency, and prevent stalls due to contention. On SandyBridge, the same coordinated multi-stage prefetching strategy is implemented, but the coordination is between the L1 software prefetches and the L2 hardware prefetcher, which is more effective than its software counterpart due to its ability to hold many more outstanding prefetch requests.

4.3 Coordinated Multi-stage Data Prefetching

As stated in Section 4.1, our coordinated multi-stage data prefetching algorithm works in three phases, namely, *what* to prefetch, *where* to insert prefetch instructions, and *when* to prefetch. For each of these phases, we discuss our specific choices for the two hardware platforms considered and reasons behind those specific choices.

4.3.1 What to Prefetch

This is the first of the three phases in which the references that should be prefetched are identified. Such references include (1) those whose future memory accesses can be determined, and (2) those when prefetched will benefit application performance.

A recent work (49) classified memory references into 5 types - (1) direct-indexed *streaming* array references, (2) direct-indexed *strided* array references, (3) *indirect*-indexed array references, (4) recursive data structures (RDS), and (5) hashing data structures. In our compiler framework, we only handle memory references of types (1) through (3), as references of types

(4) and (5) need to be handled differently.

Further, even among references whose future access patterns can be statically determined, prefetching all of them is not always beneficial. For example, references that have temporal locality in the inner loops such as references A and C in *matmul* (shown in Figure 4.2) have small reuse distances and thus, the data referenced by them stays in the cache with a high probability. Such references occur often in SPEC benchmarks and lead to redundant prefetches that reduce performance gains, especially when the amount of computation in loop-nests is small (as in *matmul*). Although Mowry et al. in (50) proposed prefetch predicates in the form of IF statements (or its equivalent) to eliminate redundant prefetches, we empirically observe that the instruction overhead of such predicates usually offset the performance gain from data prefetching. These predicates also hinder automatic vectorization by the compiler. It is for these reasons that we do not consider such references as profitable candidates for prefetching. In addition, for references that have group reuse, such as $CV[i][j]$ and $CV[i][[j+1]$ in statement S1 in *swim* (shown in Figure 4.1), we prefetch only for the *leading* reference $CV[i][j+1]$ as it is the one that accesses new data first and caches it for later use.

```
for(i = 0; i < N; i++)
  for(k = 0;  k< N; k++)
    for(j = 0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j];
```

Figure 4.2: The *matmul* kernel.

For all references that are thus marked for prefetching, our prefetching algorithm inserts prefetch instructions for just the L1 cache in case of SandyBridge since it relies on the hardware prefetcher for prefetching data to other levels. In case of Xeon Phi, however, the algorithm inserts prefetch instructions for all levels of cache to implement pure software-directed coordinated multi-stage prefetching.

4.3.2 Where to Insert Prefetch Instructions

Having identified the array references to be prefetched, the next phase is to determine *where* prefetch instructions for the identified references should be inserted. For all identified references, we insert prefetch requests in the innermost loop. This simplifies the insertion of prefetch instructions and prefetch distance calculation by the compiler. However, even in the innermost loop, the placement of prefetch instructions is important. For example, in the loop nest from the *swim* benchmark shown in Figure 4.1, if prefetch requests for all 14 array references are placed contiguously in the source code without any intermittent computation, then it might lead to pipeline-stalls because prefetches will be blocked waiting for the availability of MSHRs. This is particularly true for L1 prefetch instructions given the small size of the L1 LFB or MSHR file. Such stalls are particularly visible on Xeon Phi where the pipeline is stalled immediately upon unavailability of MSHRs, whereas incoming requests could be buffered in the load buffer in case of dynamically scheduled SandyBridge processor. Thus, our compiler framework inserts prefetch requests *between* individual statements requesting data needed by the array references in that statement - this introduces computation between a batch of prefetch requests providing adequate time for them to finish and free MSHRs. Further, prefetch requests for different levels of cache are intermingled, creating additional cycles for an L1 prefetch request to prefetch data from a lower-level cache.

4.3.3 When to Prefetch

This is the last phase of our prefetching algorithm that determines *when* to prefetch data for a memory reference. In particular, this phase determines for each array reference, the prefetch distance to be used when prefetching data to a particular level of cache. Prefetch distance calculation involves calculation of the loop iteration time, which is hard to precisely determine for processors with out-of-order execution capability. Thus, in this section, we treat the prefetch distance calculation for the in-order many-core Xeon Phi and the out-of-order multi-core SandyBridge separately.

Calculating Prefetch Distance in Xeon Phi

Mowry et al. in one of the earliest works on software prefetching (50) defined prefetch distance as $\lceil \frac{Lat}{LIT} \rceil$, where Lat and LIT are the prefetch **Lat**ency and the **L**oop **I**teration **T**ime, respectively. Of the two parameters, Lat is a machine specific parameter and can be known from vendor's data sheets or measurements. LIT, on the other hand, must be estimated by the compiler. In our framework, we estimate LIT using Equation 4.1 below. This estimated value of LIT is then used to calculate the prefetch distances for prefetches to any level of cache, using measures of the corresponding prefetch latencies.

$$LIT = nrefs * Lat_{L1} + nrefs * C_p * i + \sum_{comp=1}^{n} C_{comp} \tag{4.1}$$

where $nrefs$ is the number of distinct array references in the loop nest, Lat_{L1} is the latency of accessing a data item in the L1 cache, C_p is the number of cycles spent in executing a prefetch instruction, and C_{comp} is the cost of one of n computations in the loop nest measured in the number of cycles.

The above formula for calculating the loop iteration time (LIT) assumes that all array references in the loop nest are accessed from the L1 cache (i.e. L1 cache hits). This is because we accomplish a multi-stage data prefetching where the data is prefetched all the way up to the L1 cache. This assumption holds for all but the initial few iterations of the loop nest, which can be considered as the warm-up phase. Also, since our framework inserts one prefetch instruction for each level of the cache memory per array reference, the instruction overhead in each loop iteration is calculated as $nrefs * C_p * i$, where i is the number of levels in the cache memory hierarchy. To facilitate further discussion in this section, we assume a 2-level cache memory hierarchy as in the Xeon Phi processor.

Once LIT is determined, the prefetch distance at each of the two levels is calculated as,

$$PD_{L1} = \left(\left\lceil \frac{Lat_{L2}}{LIT} \right\rceil\right) * \alpha * \beta \tag{4.2}$$

and

$$PD_{L2} = \left(\left\lceil \frac{Lat_{memory}}{LIT} \right\rceil\right) * \alpha * \beta \tag{4.3}$$

where α and β are program dependent constants as explained below

When calculating the prefetch distance, it is important to consider whether or not the innermost loop is vectorized. This is because, the prefetch distance computed using the formula, $\lceil \frac{Lat}{LIT} \rceil$, gives the prefetch distance in the number of loop iterations. However, if the innermost loop is vectorized, the same formula (using appropriate computation costs for vector operations) gives the prefetch distance in the number of vector iterations. We accommodate this in our algorithm by the parameter α, where α is size of the vector (in the number of data elements) when the innermost loop is vectorized and 1, otherwise. The prefetch distance is further multiplied by the constant stride, β, for an array reference that exhibits strided access pattern. The prefetch distance is also rounded to the next higher multiple of the cache line size in cases when the innermost loop is not vectorized.

From Equations 4.2 and 4.3, we observe PD_{L2} is much larger than PD_{L1} (as $Lat_{memory} >> Lat_{L2}$). As a result, data is prefetched first to the larger L2 cache using a larger prefetch distance, and this data is then prefetched to the L1 cache a fewer iterations ahead. Thus, the 2 stages of data prefetching coordinate to timely fetch the data to the L1 cache. As discussed earlier, using a smaller L1 prefetch distance facilitated by this strategy minimizes contention (and also prevents a possible cache pollution due to early prefetches).

Calculating Prefetch Distance in SandyBridge

Unlike the many-core Xeon Phi that has single-wide in-order issue and in-order execution, the multi-cores are usually superscalar processors with dynamic scheduling capability. As a result, it is non-trivial to statically determine the iteration time of loops on a multi-core processor. It is for this reason that Lee et al. (49) use the IPC values from benchmark profiling in their prefetch distance calculation. Their calculation gives a lower bound on the prefetch distance. The prefetch distance can, however, be increased without any negative effects until the newly prefetched data begins to replace the previously prefetched but unused data. A larger prefetch distance for L1 prefetches is, infact, required as this helps to increase the prefetch distance for the hardware prefetcher. This, as explained earlier, helps to prevent stalls at page boundaries and in particular, helps small loops that need large prefetch distances. In our algorithm we prevent the replacement of useful data from the L1 cache by ensuring that the amount of unused prefetched data does not exceed a fourth of the L1 cache size. We restrict ourselves to a fourth of the L1 cache size as misses due to cache interference can set in much before the cache capacity is reached. The prefetch distance is thus calculated as

$$PD_{L1} = \left\lceil \frac{Size_{L1}}{4 * nrefs * Size_{elem}} \right\rceil \tag{4.4}$$

where $Size_{L1}$ is the L1 cache size, and $Size_{elem}$ is the size of each data element. The L1 software prefetch instructions use this prefetch distance and in turn trigger the L2 hardware prefetcher which in time runs sufficiently ahead of the L1 prefetches to timely bring the

```
for (iT=0; iT< N/I; iT++)
  for (kT=0; kT< N/K; kT++)
    for (jT=0; jT< N/J; jT++)
      for  (i = I*iT ;  i < I*iT + I-1 ; i++)
        for  (k = K*kT ;  k < K*kT + K-1 ; k++)
          for  (j = J*jT ;  j < J*JT + J-1 ; j++)
            {
            C[i][j] += A[i][k] * B[k][j];
            }
```

(a)

```
for (iT=0; iT< N/I; iT++)
  for (kT=0; kT< N/K; kT++)
    for (jT=0; jT< N/J; jT++)
      for  (i = I*iT ;  i < I*iT + I-1 ; i++)
        for  (k = K*kT ;  k < K*kT + K-1 ; k++)
          for  (j = J*jT ;  j < J*JT + J-1 ; j++)  {
            prefetch (&B[k][j+PDL1], _L1_);
            prefetch (&B[k][j+PDL2], _L2_);
           C[i][j] += A[i][k] * B[k][j];  }
```

(b)

```
for (iT=0; iT< N/I; iT++)
  for (kT=0; kT< N/K; kT++)
    for (jT=0; jT< N/J; jT++)
      for  (i = I*iT ;  i < I*iT + I-1 ; i++)
        for  (k = K*kT ;  k < K*kT + K-1 ; k++)
          for  (j = J*jT ;  j < J*JT + J-1 ; j++)  {
            prefetch (&B[k+1][j], _L1_);
            prefetch (&B[k+2][j], _L2_);
           C[i][j] += A[i][k] * B[k][j];  }
```

(c)

Figure 4.3: (a) Tiled *matmul*; (b) Tiled *matmul* with prefetching as in the *general* case; (c) Tiled *matmul* with prefetching as in the *special* case.

data to the L2 cache. Thus, like Xeon Phi, coordinated multi-stage prefetching that prefetches the data all the way to the L1 cache with minimal contention in L1 LFB, is accomplished on SandyBridge.

Prefetch Distance for Tiled Code

While the formula for calculating the prefetch distances as derived above proves applicable to most programs, it nevertheless assumes long contiguous streaming/strided access patterns, i.e. array references where the entire array is referenced in all loop nests within an application. Such access patterns are not observed, however, in tiled codes and certain other scientific codes where the computational domain is split into parts, as in *gemsfdtd* and *mgrid* benchmarks from the SPEC suite.

We use a simple example of tiled *matmul* to illustrate such a case and the prefetching strategy applied therein. Figure 4.3(b) shows the tiled *matmul* code with prefetch instructions inserted assuming contiguous streaming access patterns - $B[k][j]$ is identified as suitable for prefetching and prefetch instructions are inserted in the innermost loop as shown. However, as a result of tiling, the access pattern is no longer contiguous - only J elements of array B are accessed contiguously, followed by another J elements from the next row of B. Thus, if a prefetch distance, PD_{L2} (that is likely to be greater than J), as calculated using Equation 4.3 is used, *useless* prefetches could result. Since the data does not arrive at L2 cache in time for L1 prefetches, 2-stage prefetching becomes ineffective.

Our framework identifies such cases of accesses to non-contiguous data chunks (where the size of the innermost loop is a fraction of the array size) as in tiled *matmul*, and inserts prefetches as shown in Figure 4.3(c). It prefetches one of the next array rows instead of prefetching elements that are a few cache lines (= prefetch distance) away. Thus, useless prefetches are reduced to only those that are prefetched while processing the last few rows of a tile of array B. Since correct data does arrive in time in L2 cache, 2-stage prefetching again becomes very effective. In such cases, the prefetch distance is calculated in the number of *array rows*, instead of elements as in the *general* case above discussed. It is given as follows.

$$PD^s_{L1} = max\left(1, \frac{PD^g_{L1}}{lc}\right) \tag{4.5}$$

and

$$PD^s_{L2} = max\left(2, \frac{PD^g_{L2}}{lc}\right) \tag{4.6}$$

where PD^s is the prefetch distance in this *special* case, PD^g is the prefetch distance in the *general* case (calculated in the number of elements) and lc is the loop count of the innermost loop, or tile size in case of tiled codes. In the formula, the lower bounds of 1 and 2 for L1 and L2 prefetch distance is to ensure sufficient time for the data to arrive in both L1 and L2 cache. In SandyBridge, our framework inserts prefetch requests for just the L1 cache using the above formula, and relies on the hardware to prefetch the data to L2 cache.

4.3.4 Multi-stage Prefetching in multi-threaded environment

Chip multiprocessing (CMP) and simultaneous multithreading (SMT) can both be used to extract the inherent instruction-level parallelism in programs run in a multithreaded environment. Both these techniques interact closely with prefetching. In CMP, threads on each core simultaneously place demands for data from the memory, and pronounced bandwidth contention results. The net effect of this is an increase in the effective memory latency. As a result, pipeline-stalls due to unavailability of slots in the load/store buffer (in multi-cores) tend to be more prominent in such cases. Data prefetch requests that are additional requests to the regular loads, also occupy slots in the load buffer and thus end up contributing to these stalls. It is for this reason that we observe that for certain memory-intensive benchmarks such as *swim*, *bwaves* and others, the performance advantage due to software-directed prefetching to L1 over the hardware prefetcher, is lost. For other benchmarks that are less memory-intensive such as *bt*, *cactus* and *matmul*, the performance advantage persists. In a many-core processor such as Xeon Phi that has blocking loads, there is no load buffer and thus data prefetch requests and regular loads do not share common resources. As a result, prefetching does not cause a side-effect to parallel performance as in multi-cores as long as useful data is timely prefetched.

In SMT, multiple SMT threads on a core share the LFB or the MSHR file. As a result, the increased data access rate puts pressure on the shared MSHR file, resulting in potential stalls due to contention. Precisely, a contention results when the maximum number of outstanding prefetches are less than the available MSHRs.

Since MSHRs are already scarce for a single thread, we observe that prefetching generally adversely affects performance of programs run in an SMT environment. In case of Xeon Phi, however, prospects of an interesting compiler optimization to reduce contention for MSHRs emerge for several reasons. Xeon Phi employs GDDR5 memory that has significantly higher latency and the latency increases when there are multiple active data streams or the number of distinct array references in a loop. It is for this reason that our framework detects contention for MSHRs and performs loop distribution prior to the insertion of prefetch instructions. This reduces the number of active streams in a loop by distributing statements in the original nest to multiple nests and thus benefits from reduced contention for MSHRs due to reduced memory latency. It is important to note that in cases where there is reuse among array references in original nest, loop distribution hurts temporal locality and thus reuse. However, since we distribute at the innermost loop, only L1 cache locality is hurt, which is more than compensated by effective data prefetching rendered possible by reduced contention. In multi-cores that employ a DDR3 memory, the latency does not increase significantly with the number of active streams, and thus loop distribution is not beneficial.

4.4 The Compiler Framework

Figure 4.4 gives an overview of our compiler framework for coordinated multi-stage data prefetching. Our prefetching strategy is implemented in the open-source ROSE (51) compiler, which performs source-to-source transformations and provides many APIs for program analysis and transformations based on its Abstract Syntax Tree (AST).

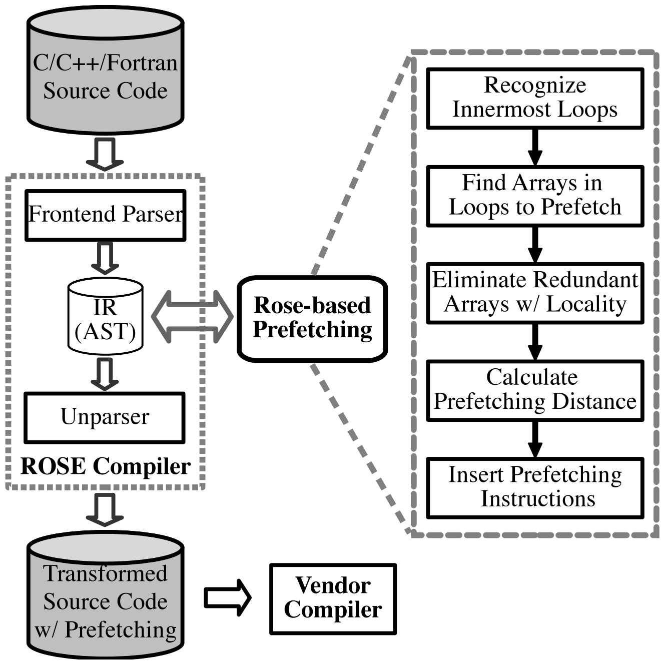

C/C++/Fortran Source Code
Frontend Parser
IR (AST)
Unparser
ROSE Compiler
Rose-based Prefetching
Recognize Innermost Loops
Find Arrays in Loops to Prefetch
Eliminate Redundant Arrays w/ Locality
Calculate Prefetching Distance
Insert Prefetching Instructions
Transformed Source Code w/ Prefetching
Vendor Compiler

Figure 4.4: Compiler framework for coordinated multi-stage prefetching.

As shown in Figure 4.4, there are mainly five steps to insert multi-stage prefetches to the input source code after it is parsed to the AST. Steps 1 through 3 determine *what* to prefetch - all array references in the innermost loops are recognized and those with either temporal or group locality are discarded for prefetching. Step 4 is the key step to determine the prefetch distances or, *when* to prefetch, for multi-stage prefetching. As stated in Section 3, prefetch distance for each cache level is calculated using Equations (2)-(6). This requires information such as the total number of array references as well as the number and type of computations involved in the

Benchmark	Benchmark Suite	Category	Problem Size
cactus	SPEC CPU2006	General Relativity	Ref Input
hmmer	SPEC CPU2006	Search Gene Sequence	Ref Input
libquantum	SPEC CPU2006	Quantum Computing	Ref Input
gemsfdtd	SPEC CPU2006	Computational Electromagnetics	Ref Input
bwaves	SPEC OMP2012	Fluid Dynamics	Ref Input
swim	SPEC OMP2012	Weather Prediction	Ref Input
mgrid	SPEC OMP2012	Fluid Dynamics	Ref Input
bt	SPEC OMP2012	Fluid Dynamics	Ref Input
matmul	BLAS	Matrix-Matrix Multiplication	N=4000
spmv	General-purpose	Sparse Matrix-Vector Multiplication	N=10000

Table 4.1: Summary of benchmarks with problem sizes.

loop nest, which are obtained by querying the AST. The last step is to insert those prefetches with the calculated prefetch distance into the AST. After our multi-stage prefetching transformation, ROSE unparses the AST to the transformed source code with prefetch instructions. Finally, a back-end vendor compiler is used to compile the transformed source code to the final executable.

4.5 Experimental Results and Discussion

In this section, we present the results of our coordinated multi-stage data prefetching and compare it with other prefetching strategies in both SandyBridge and Xeon Phi processors. Results comparing multi-stage prefetching and other strategies in a multithreaded environment are also presented.

4.5.1 Experimental Environment

We first present our experimental setup. We chose a diverse set of memory-intensive benchmarks from the SPEC CPU2006 and OMP2012 benchmark suites that have high cache miss rates. We also chose two frequently used kernels, *dense matrix-matrix multiplication* (that can be tiled) and *sparse matrix-vector multiplication* (that involves indirect array indexing). Table 4.1 lists all the benchmarks used with their problem sizes.

We ran our experiments on the two different hardware platforms discussed in this chapter - the multi-core Intel SandyBridge and the recently released many-core Intel Xeon Phi. The micro-architectural details of the two processors are compared in Table 4.2.

Also, Xeon Phi contains instructions for prefetching data as ‘exclusive’ into either of the two caches, whereas SandyBridge does not provide this functionality. On Xeon Phi, we use the 'exclusive' prefetches to prefetch writes. Since Xeon Phi has 2-level cache hierarchy and Sandy-Bridge (with 3-level cache hierarchy) also allows to prefetch only to the L1 and L2 caches, the coordinated prefetching strategy is implemented in **2** stages on both platforms. After generating the transformed source code that contains the prefetch instructions, we use the Intel compiler (v13) (52) as the back-end compiler to generate the executable.

Microarchitecture	Cache	No. of Cores	SMT	Hardware prefetcher
SandyBridge	Non-blocking	8	2-way	Streaming, Strided (L1+L2)
Xeon Phi	Blocking	60	4-way	Streaming (L2 only)

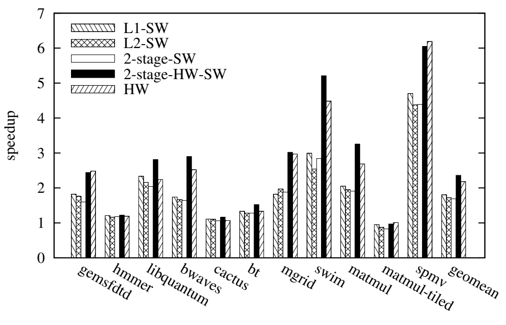
Table 4.2: Details of the microarchitectures

4.5.2 Results for Multi-stage Prefetching in single-threaded environment

Figure 4.5 compares the performance results obtained on SandyBridge and Xeon Phi, respectively, for the different prefetching strategies summarized in Table 4.3. In addition to the inherent hardware-based prefetching on Intel processors and the icc-directed prefetching, we implemented four other prefetching strategies for the purpose of comparison. As discussed, we propose a 2-stage coordinated software prefetching for Xeon Phi and 2-stage coordinated hardware-software prefetching for SandyBridge. Of the seven different strategies listed in Table 4.3, we could not implement the baseline prefetching on Xeon Phi because the BIOS did not provide the facility to turn off the hardware prefetcher as in SandyBridge. Thus, the hardware-based prefetching serves as the baseline in case of Xeon Phi. On the other hand, the Intel compiler for multi-cores is largely passive in the matter of prefetching and relies primarily on

Strategy	Description
Hardware prefetching	The prefetching strategy used by Intel processors
L1 SW pref.	Data is prefetched only to L1 cache
L2 SW pref.	Data is prefetched only to L2 cache
2-stage SW pref.	Data is prefetched to both L1 and L2 cache using carefully chosen prefetch distances
2-stage HW-SW pref.	Data is prefetched to L1 cache assuming that hardware prefetcher brings the data to the L2 cache
icc SW pref.	The prefetching strategy used by the Intel compiler
Baseline pref.	The baseline configuration with all prefetching disabled

Table 4.3: Summary of different prefetching strategies.



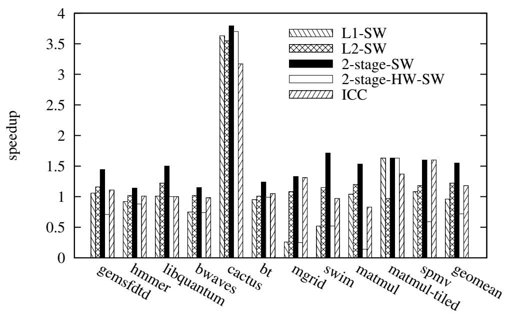


Figure 4.5: Performance speedup of different prefetching strategies on single thread in (a) SandyBridge (results normalized wrt baseline pref.) (b) Xeon Phi (results normalized wrt hardware pref.)

the hardware prefetcher. Thus, in our results for SandyBridge, we do not show the performance results for icc-directed prefetching which performs similar to the hardware-based prefetching for all benchmarks. In our analysis, we explain the behavior of the prefetching strategy adopted by the Intel compiler by studying the assembly code generated for different benchmarks.

According to the data access pattern, we divide the benchmarks into four categories: 1) programs with streaming or strided accesses, 2) programs with indirect array indexes, 3) programs with array of structures (and pointers), and 4) programs with loop sizes that are fractions of the problem size.

1. Programs with streaming or strided accesses. Among the benchmarks used for our experiments, *cactus*, *hmmer*, *bwaves*, *swim*, and *matmul* involve streaming array accesses whereas *bt* involves streaming as well as strided accesses. Results show that on both hardware platforms, 2-stage coordinated prefetching as proposed for the respective platform, either outperforms or performs close to the best performing prefetching strategy for all benchmarks. It is interesting to note that 2-stage coordinated hw-sw prefetching which is the best performing strategy on SandyBridge, is the worst performing on Xeon Phi owing to the difference in interaction between hardware and software prefetching on the two platforms. On Xeon Phi, the software prefetches cannot train the hardware prefetcher and both cannot co-exist. On SandyBridge, on the other hand, we employ software-based prefetching to train the hardware prefetcher (because such an opportunity is provided by hardware) to overcome its limitations, and thus outperform other strategies. It is similarly interesting that the 2-stage sw prefetching which is the best performing strategy on Xeon Phi performs poorly on SandyBridge. This is because of a very small

LFB at the L2 cache in SandyBridge. The L2 sw prefetching performs poorly on SandyBridge for the same reason. The L1 sw prefetching performs worse than the L2 sw prefetching on Xeon Phi due to fewer MSHRs at L1 cache (and thus more contention), whereas it performs slightly better than the L2 sw prefetching on SandyBridge due to prefetching the data to the L1 cache (that has same-sized LFB as L2 cache). The performance results for each benchmark are discussed in more detail below.

swim, bwaves and matmul. These benchmarks involve streaming access patterns with long loops. Although *matmul* has temporal locality, it behaves like other streaming benchmarks when it is not tiled since the data set size is much larger than L2 cache size. Here, multi-stage prefetching performs considerably better than the other prefetching strategies. It wins over the hardware and L2 sw prefetching for hiding the L2-to-L1 latency by prefetching the data to the L1 cache. The improvement is significant since the L1 misses are significant given the streaming nature and large working sets in these benchmarks. On SandyBridge, the 2-stage coordinated hw-sw prefetching improves considerably over the hardware-based prefetching in case of *matmul* even though the L1 hardware prefetcher proves sufficient for the small number of streams in *matmul*. This is because L1 software prefetches with large prefetch distances help in increasing the prefetch distance of the hardware prefetcher that is essential for the small loops in *matmul*. The L1 hardware prefetcher on SandyBridge prefetches only the next cache line, and thus cannot sufficiently increase the prefetch distance of the L2 hardware prefetcher. It is for this reason that we observe (using the Vtune performance monitoring tool (53)) that a significantly more number of loads that miss the L1 cache hit the LFB, in case of *matmul* that uses the hardware prefetcher against the one which uses our 2-stage coordinated software prefetching. The higher number for the former suggests pending prefetch requests at the L2 cache because of insufficient prefetch distance. On Xeon Phi, the Intel compiler performs worse than even the hardware prefetcher in *matmul*, because of inserting redundant prefetch instructions for the array reference (reference $C[i][j]$ in Figure 4.2) that has temporal locality in an inner loop. In *swim*, the Intel compiler performs poorly as it does not prefetch data for all references. In addition, the data is prefetched only to the L2 cache and not to the L1 cache. In *bwaves*, the Intel compiler performs worse than coordinated prefetching because of issuing redundant prefetches for references that have temporal locality in the innermost loop, while leaving out references that have spatial locality in the innermost loop. We believe that it makes a wrong decision because of performing loop interchange optimization. It also does not prefetch data to the L1 cache in this case.

cactus. In *cactus*, bulk of the computation and memory references happen in a very large loop nest. Each iteration of the loop nest provides sufficient cycles to hide the memory latency. Since the loop nest involves significant computation, even requests to prefetch the data directly to the L1 cache are finished without much stalls. As a result, all software prefetching strategies perform similarly. The multi-stage prefetching achieves slightly better performance as compared to the L1 or L2 prefetching because of achieving better performance in other smaller loop nests that also involve streaming accesses. On Xeon Phi, the Intel compiler performs slightly worse because of inserting all prefetch instructions at the end of the loop instead of interleaving them with the computation. This leads to stalls due to contention at L1 cache that hurts performance. An important observation here is that the baseline hardware prefetcher on Xeon Phi performs significantly worse than the other strategies. This is because, the large loop nest contains streams (81 data streams) that are much larger than that can be handled by the hardware prefetcher (16, in case of Intel Xeon Phi). The performance difference is much less on SandyBridge for 2 reasons, (1) the hardware prefetcher can prefetch data for 32 streams instead of 16, and (2) the out-of-order execution tolerates most of the stalls since there is significant computation interspersing memory requests in the large loop nest in cactus.

hmmer. In *hmmer*, a subroutine called *P7Viterbi* is most computationally intensive and contains small loops. All the data referenced in the subroutine fits the L2 cache. Thus, although 2-stage prefetching that brings the data to the L1 cache wins over other strategies, the performance difference is small. On Xeon Phi, the Intel compiler also prefetches to both the L1 and L2 cache, but uses a much larger prefetch distance which proves costly, given the small loops.

bt. In *bt*, four subroutines, *compute_rhs*, *x_solve*, *y_solve* and *z_solve* contribute almost entirely to the execution time. These subroutines contain both streaming as well as strided accesses, and the working set is large. Thus, although a likely candidate for significant performance gains from 2-stage prefetching as in *swim*, the performance gains are small. This is because, the loop nests in *bt*, particularly some of the time-consuming nests in *x_solve*, *y_solve* and *z_solve*, are dominated by computation than by memory references. The performance improvement achieved by coordinated prefetching on SandyBridge is smaller than that achieved on Xeon Phi because higher computation in loop nests allows for tolerance of stalls due to out-of-order execution. Most of the performance improvement achieved by 2-stage prefetching stems from prefetching for strided accesses. These strided accesses are missed by even the strided

```
void quantum_cnot( quantum_reg *reg, . . . )
  . . .
   for (i = 0; i < reg->size; i++)
       prefetch(&reg->node[i+PDL1], _L1_E );  // exclusive
       prefetch(&reg->node[i+PDL2], _L2_E );
       if (reg->node[i].state &  . . .  )
           reg->node[i].state ^= . . .
   . . .
```

Figure 4.6: An example loop nest with prefetching instructions to L1 and L2 cache in the *libquantum* benchmark.

hardware prefetcher in SandyBridge (that can detect strides upto 2K bytes) since the strides are long, given loops with large trip count. On Xeon Phi, the Intel compiler does not yield much improvement over the baseline because of not adequately prefetching for strided accesses.

2. Programs with array of structures (and pointers). In *libquantum*, the computationally intensive loop-nests contain an array of structures, that is responsible for bulk of the memory accesses. One such loop-nest is shown in Figure 4.6, and the memory reference $reg \rightarrow node$ is an array of structures, that has $state$ as one of its fields. In such cases, our multi-stage prefetching algorithm determines the size of the structure and prefetches the entire structure on the assumption that majority of the fields of the structure will be referenced - this may lead to prefetching more than a single cache line. However, in *libquantum*, the structure has only 2 fields and a size of 16 bytes, so we prefetch just one cache line. Our multi-stage prefetching wins over the hardware-based prefetching, as the hardware prefetcher cannot run sufficiently ahead of the program counter for timely prefetching, given the small computation in the loop-nests and larger size of the data structure. It is for this reason that on SandyBridge, coordinated hw-sw prefetching where L1 software prefetches train and thus help in increasing the prefetch distance of the hardware prefetcher outperforms all other strategies. The performance improvement of coordinated sw prefetching over the baseline hardware prefetcher on Xeon Phi is more significant since there is no hardware prefetcher at L1 in Xeon Phi to prefetch to the L1 cache. On Xeon Phi, the Intel compiler does not prefetch array of structures and thus performs as well as the hardware prefetcher.

3. Programs with indirect array indexes. An example program with indirect array indexes is *sparse matrix-vector multiplication* as shown in Figure 4.7, where the array reference

```
for (i=0; i<M; i++)
  int start = row_start[i], stop = row_start[i+1];
  for (j=start; j<stop; j++)
     prefetch (&values[j+PDL1], _L1_ );
     prefetch (&values[j+PDL2], _L2_ );
     prefetch (&colIdx[j+PDL1], _L1_ );
     prefetch (&colIdx[j+PDL2], _L2_ );
     C[i] += values[j] + B[colIdx[j]];
```

Figure 4.7: *Sparse matrix-vector multiplication* with prefetching instructions to L1 and L2 cache.

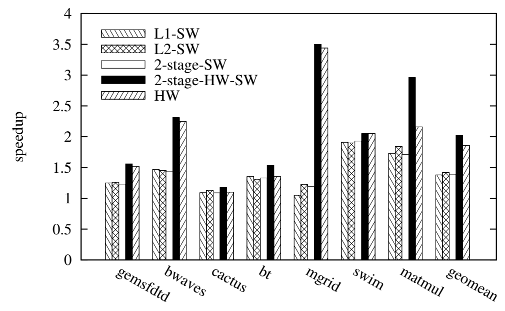
$B[colIdx[j]]$ is indirectly indexed. In such cases, we prefetch data for the directly indexed reference, $colIdx$, that is used to reference the indirectly indexed reference B. We also prefetch the array reference $values$, but not reference C that has temporal locality in the innermost loop j. On Xeon Phi, we get considerable performance improvement over the baseline hardware prefetcher due to prefetching to L1 cache. On SandyBridge, however, the hardware prefetcher performs slightly better than the 2-stage coordinated hw-sw prefetching as even the hardware prefetcher at L1 cache is successful in bringing the data to the L1 cache given few memory references in the loop nest. The slight better performance of the hardware prefetcher is due to no instruction overhead of prefetching. It is important to note that the loop iteration time in *spmv* is not as small as in *matmul* due to an indirectly indexed array reference that hurts efficient vectorization, and thus the benefit from increasing the prefetch distance of the hardware prefetcher through software prefetches is not significant. On Xeon Phi, the Intel compiler performs similar to the coordinated sw prefetching since it also prefetches to both the L1 and L2 cache using different distances. Although it uses a much larger distance that needed for the L1 prefetches, the performance is not hurt due to very long data streams.

4. Programs with loop sizes that are fractions of the problem size. This category of programs include both the tiled codes and various scientific codes that compute in parts such as the *gemsfdtd* and *mgrid* benchmarks from SPEC suite. In such codes, our multi-stage prefetching algorithm prefetches data in the following array rows instead of prefetching a few cache lines ahead as shown earlier in Figure 4.3(c). This helps to not only avoid prefetching useless data but also allows to timely prefetch useful data. On Xeon Phi, coordinated software prefetching wins over the hardware prefetcher and other software prefetching strategies primarily due to prefetching the data to L1 cache and also eliminating redundant prefetches. On SandyBridge, hardware-based prefetching and coordinated hw-sw prefetching win over others due to employing an efficient hardware prefetcher (with no resource contention) to prefetch data to the L2 cache. Other details regarding performance results for the 3 benchmarks in this category are discussed as follows.

gemsfdtd. On SandyBridge, the coordinated hardware-software prefetching performs slightly worse than the hardware prefetcher because although this benchmark computes in 2 parts, most of the execution belongs to one of those 2 parts. As a result, aggressive prefetching by hardware enables timely prefetching for the 2 parts although at the cost of some useless prefetches. The overhead due to both the useless prefetches and not prefetching the data to the L1 cache are tolerated by the out-of-order execution since the loops nests are not highly memory-intensive and involve considerable computation. On Xeon Phi, the Intel compiler does not achieve improvement over the hardware prefetcher due to insertion of useless prefetches by prefetching using large prefetch distances.

mgrid. On SandyBridge, the improvement achieved by coordinated hw-sw prefetching over the hardware prefetcher is small because this benchmark is highly memory intensive, and thus we observe significant pipeline stalls due to filled up load buffer, when prefetch requests are inserted. Using Vtune, we observe that the stalls due to filled up load buffer in a code with L1 software prefetch instructions increase by a factor of 3 over a code that contains no prefetch instructions. On Xeon Phi, however, there are no additional stalls due to prefetching and thus significant improvement is obtained over the hardware prefetcher. The Intel compiler in this case inserts prefetches for both the L1 and L2 caches using different distances. The L1 prefetch distance is again larger than needed but does not hurt performance since this benchmark is less sensitive to prefetch distance.

matmul-tiled. We tile *matmul* using the same tile size as chosen by *icc*, i.e. 128. As a result of using a tile size of 128, the working is reduced such that it exceeds the L1 cache size, but is well within the L2 cache size. Thus, a single tile always fits the L2 cache, and once fetched, the data needs to only be prefetched from the L2 cache in subsequent executions of the tile. This gives interesting performance results on Xeon Phi - all prefetching strategies that prefetch the data to the L1 cache perform as well as the coordinated sw prefetching. However, *icc* performs slightly worse because of inserting redundant prefetches for references with locality in the inner loops as in *matmul*. On SandyBridge, the L1 hardware prefetcher effectively prefetches data to the L1 cache, given few references involved, and thus performs slightly better than coordinated



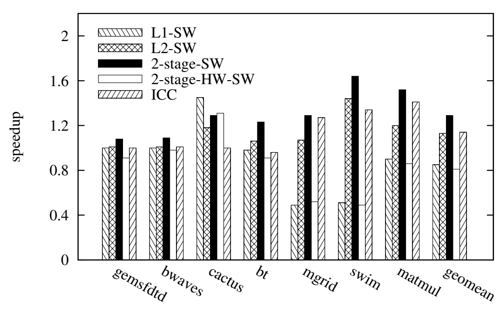


Figure 4.8: Performance speedup of different prefetching strategies using CMP technique in (a) SandyBridge (results normalized wrt baseline pref.) (b) Xeon Phi (results normalized wrt hardware pref.)

hw-sw prefetching. The other strategies only perform slightly worse since the working set occupies the L2 cache, and part of the L2-L1 latency can be tolerated through out-of-order execution.

4.5.3 Results for Multi-stage Prefetching in multithreaded environment

Figure 4.8 shows the performance results of our coordinated prefetching strategies for Sandy-Bridge and Xeon Phi on 8 and 32 cores, respectively, in CMP environment. On SandyBridge, as discussed in Section 4.3.4, the performance improvements achieved by coordinated hw-sw prefetching using single thread diminish for benchmarks that are highly memory-intensive due to increased stalls from resource contention. The benchmarks in this list are *swim*, *gemsfdtd*, *bwaves* and *mgrid*. Other less memory intensive benchmarks such as *bt* and *cactus* continue to show improvement. *Matmul*, although memory intensive, shows significant improvement owing to the increase in hardware prefetch distance achieved through software prefetching. On Xeon Phi, the benchmarks behave similar in both the CMP and single thread environment since prefetching does not cause additional stalls in the CMP environment as in SandyBridge.

As discussed in Section 4.3.4, memory intensive benchmarks with many references in the loop nest benefit from loop distribution optimization on Xeon Phi on account of reduced contention. Figure 4.9 shows the performance gains of loop distribution (followed by multi-stage prefetching) in 3 memory-intensive parallel benchmarks, *swim*, *gemsfdtd* and *cactus*. Using the recommended 2-way SMT on Xeon Phi, loop distribution achieves another 13% average

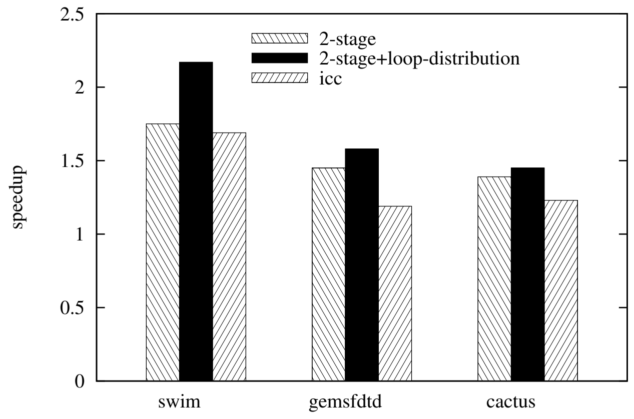


Figure 4.9: Performance speedup with loop distribution in Xeon Phi (results normalized wrt hardware pref.)

speedup over multi-stage prefetching on the 3 benchmarks.

4.6 Related Work

In the past, there has been significant research on tuning the prefetch distance and reducing overhead due to software prefetching (54; 55; 56; 50). Mowry et al. (50) defined prefetch distance in terms of the prefetch latency and loop iteration time. Mowry et al. also proposed prefetch predicates to eliminate redundant prefetches. Badawy et al. in (54) proposed to insert prefetch instructions in the epilogue and prologue. Recently, Lee et al. (49) have proposed to calculate the loop iteration time using the average IPC of the application obtained through program profiling. Such an adjustment to prefetch distance is necessary for modern multi-cores that execute out-of-order. In our work, however, since we only use software to prefetch to the L1 cache in multi-cores, we show that using a large prefetch distance such that it does not cause evictions in the L1 cache is a better choice for several reasons. This has the advantage of being loop-based and not program-based, and is calculated statically.

In recent past, coordinated prefetching was proposed and implemented in IBM's Power6 microarchitecture (45). On Power6 (and its follow-ons), the L2 hardware prefetcher that can hold 32 outstanding prefetch requests, prefetches data from memory using a larger distance (at most 24 lines) and the L1 hardware prefetcher that can hold 8 prefetch requests, prefetches data from L1 using a smaller distance (at most 2 lines). In this work, we use a similar idea to achieve coordination, but through software prefetching. On SandyBridge, software prefetching at L1 coordinates with the L2 hardware prefetcher that allows us to significantly increase the prefetch distance (without polluting the cache - Section 4.3.3) and overcome the limitations of the hardware prefetcher as mentioned in Section 4.1. On Xeon Phi, the coordination is achieved entirely through software prefetching. Lee et al. (49) have considered coordination between hardware and software prefetching on existing multi-cores. They manually insert software prefetch instructions and identify benchmarks for their positive, neutral or negative interaction with the existing hardware prefetchers (primarily through simulation). Their work is thus focused on studying the interaction between hardware and software prefetching, but does not propose any particular prefetching strategy. In their tests, they use cooperative hardware-software prefetching where both software and hardware prefetching co-exist; ours is a coordinated hardware-software prefetching technique, where the software prefetches actively train the hardware prefetcher to achieve coordination.

In the past, Santhanam et al. (57) and Caragea et al. (58) have also proposed reduction of prefetch distance due to limited resources in the form of number of outstanding prefetch requests that can be handled by the hardware. However, these works consider HP PA-8000 and a many-core research machine, XMT, respectively, both of which employ single-level cache. Thus, unlike our work, they do not employ the facility to prefetch to multiple levels of cache as in existing architectures, to tackle the problem of resource contention. A recent work from Intel (59) talks about prefetching the data to both the L1 and L2 cache - a technique implemented in the existing version of the Intel compiler for Xeon Phi. However, they do not describe any strategy to prefetch the data to the L1 and L2 cache such that a coordination is achieved. We observe in our results that the Intel compiler for Xeon Phi does not always insert prefetch requests for L1 and L2 cache, and generally uses a larger prefetch distance at L1 than needed. As a result, it cannot match the performance of our coordinated prefetching on Xeon Phi.

Among other works in software-based prefetching are those that employ helper threads to predict future load addresses (60; 61; 62). The authors in (60; 61) use an idle thread as the helper thread to prefetch data for another compute thread, whereas Son et al. (62) extend the helper-thread prefetching to work with multiple cores by assigning a customized helper thread to a group of compute threads. These works, however, have not evaluated the impact of resource contention due to aggressive prefetching by the helper-thread. In our experiments, we find the impact of resource contention to be important, particularly on many-cores that have blocking caches, and thus implement a simple and low-overhead software prefetching strategy that is different from helper-thread based prefetching.

4.7 Conclusion

In this work, we study software prefetching in light of various architectural features that support prefetching in existing processors. Based on our study of those features, we implement a coordinated multi-stage prefetching strategy for two widely different state-of-the-art processors, the multi-core SandyBridge and the many-core Xeon Phi. However, the means of achieving this coordination is different on either - in SandyBridge, it is achieved through the MLC hardware prefetcher prefetching data to the L2 cache and L1 software prefetches further bringing it to L1 cache; in Xeon Phi, the coordination is achieved through carefully tuned software prefetching at different levels of cache. Results establish the efficacy of these strategies on respective platforms and also the importance of an awareness of the influence of different architectural features on prefetching as studied in this work. The performance results achieved using the 7 different prefetching strategies are discussed for each benchmark considered. Further, the performance of these strategies in multithreaded environment is also presented and discussed. Since our multi-stage coordinated prefetching is a simple, static prefetching technique, and does not require any special hardware, it can be readily incorporated in production-quality compilers for existing architectures.

Chapter 5

Loop Fusion and Real Applications - Part 1

5.1 Introduction

With the increase in the number of cores on chip (or, processors in a node) and the consequent accentuation of the existing problems of *memory* and *bandwidth wall*, there is a renewed focus on the optimization capabilities of a parallelizing compiler. A parallelizing compiler should not only help to exploit the available parallelism in the host hardware, but also alleviate the problems of memory and bandwidth wall by performing memory optimizations such as *loop tiling, data prefetching, loop fusion* and other supporting optimizations such as *loop shifting* and *loop interchange*. While these important responsibilities of a parallelizing compiler are well recognized, it is also well known that compilers often fall way short in capitalizing on the optimization opportunities provided by a target application.

One key reason for this shortfall is that most existing production compilers built upon traditional wisdom, often limit themselves to optimizing only small scopes in the entire program (63; 64; 65; 66) or kernels (6; 67; 68; 69). Even in some recent work, such as (70), the authors only analyze individual hot loops in target applications to mark those loops as parallel or determine legality of loop distribution. However, our experiments with several scientific applications from the SPEC benchmark Suite reveal that there are many opportunities for improvement in memory (and parallel) performance of those benchmarks through global program optimizations (or transformations) such as applying loop fusion across a sequence of such hot loops. Such loop fusion across multiple loop nests saves the cost of fork-join synchronization between loops, and more importantly, significantly improves *temporal locality* and thus saves costly off-chip memory accesses. Both these benefits are of increasing importance in the era of multi-/many-cores.

In the past, there has been work on loop fusion (71; 72; 73; 65), but the existing production compilers still prove insufficient in fusing multiple loop nests. We find that, in addition to some of the other limitations such as non-conformable loop bounds or loop orders in different nests, and existence of imperfect nests, it is the existence of (extra-)stringent memory dependences on temporary variables (scalars or low-dimensional arrays as shown in Figure 5.1a) in different nests that is the key factor behind the dismal performance of existing compilers with regards to global program optimization. Such memory dependences severely limit the degrees of freedom for loop transformations such as loop fusion and loop interchange. It is important to note that we cannot simply get rid of these dependences to enable loop fusion, because fusion merges multiple *definitions* and *uses* in the fused loop's body that can potentially violate program correctness. Such dependences thus require specific treatment. In this work, we make the following contributions,

1. We analyze an important cause of the incapability of the existing compilers in achieving global program transformations - the (extra-)stringent data dependences caused by temporary variables that appear in different loop nests and have the same variable-name.

2. We propose *variable liberalization*[1], a technique that strategically relaxes (or liberalizes) dependences on temporary variables in different nests so as to release the degrees of freedom needed to express effective loop transformations such as loop fusion and loop interchange while preserving program correctness. *Variable liberalization* is different from *variable privatization* because it relaxes dependences on multiple outer loops across loop-nests instead of just the outer loop in a single nest, and it is performed to also allow fusion in addition to just parallelization. *Liberalization* is different from *expansion* because there is no real expansion of variable dimensions that takes place. It can thus be seen as a combination of both *privatization* and *expansion*, drawing the benefits of both.

[1]Liberalization literally means relaxing previously existing rules to, for example, allow freer interaction between 2 parties. Variable liberalization similarly relaxes the extra-stringent data dependences caused by temporary variables in 2 loop-nests to allow global optimizations such as loop fusion between them.

3. We implement our work in the state-of-the-art PLuTo polyhedral compiler and evaluate the improvement obtained in terms of parallel performance on 8 hot regions in 5 subroutines of 4 real scientific applications from the SPEC and NAS Parallel Benchmark Suites that have not been effectively optimized by current production compilers even though they contain significant opportunity. In addition, as a result of pruning/relaxing dependences to achieve *variable liberalization*, we significantly reduce the memory requirement and the compile time of large application programs as compared to state-of-the art polyhedral compilers, thus addressing the scalability problem within such compilers to some extent.

The rest of the chapter is organized as follows. Section 5.2 reinforces the motivation for this work through a simple example program that exposes the limitation of existing compilers in global program transformation, and shows the potential benefits from overcoming that limitation. Section 5.3 provides a background on dependence analysis with focus on temporary variables, and also introduces key concepts for the understanding of our proposed *variable liberalization* optimization. This is followed by a detailed discussion of our approach, and its application in the different cases seen in real applications in Section 5.4. Section 5.5 puts all previous discussion together in the form of an algorithm that implements *liberalization*. In Section 5.6, we evaluate our approach against state-of-the-art compilers and discuss results in each case. The related work is presented in Section 5.7. Finally, Section 5.8 concludes our work.

5.2 Motivation

Figure 5.1a shows an example program that assumes some of the features, characteristic of real scientific applications. These features include, extensive use of temporary variables such as the scalar variable a and the array variable tmp, excellent opportunities for data reuse across loop nests such as that in arrays rho, x and z, and imperfectly nested loops. Figure 5.1b shows a transformed program where the two loop nests in the original program (Figure 5.1a) are fused into a single nest and the outermost loop has been marked parallel (after marking variables a and tmp as *private* to each thread) . Clearly, the transformed program is equivalent to the original program since all memory dependences are satisfied. Figure 5.1c shows the transformed program as generated by PLuTo using its most effective fusion heuristic, *smartfuse*.

Table 5.1 shows the sequential and parallel performance of the two transformed programs normalized with respect to the performance achieved by the Intel Compiler (ICC with '-O3

```
for(i1=0;i1<Nx;i1++) {
 for(j1=0;j1<Ny;j1++) {

  for(k1=0;k1<Nz;k1++) {
   S1: a = rho[i1][j1][k1];
   S2: tmp[k1] = a*b + c;
   S3: x[i1][j1][k1] = x[i1][j1][k1] + a*c; }

  for(k1=1;k1<Nz-1;k1++) {
   S4: z[i1][j1][k1] = x[i1][j1][k1] + tmp[k1+1] - tmp[k1-1]; }

} }

for(i2=0;i2<Nx;i2++) {
 for(j2=0;j2<Ny;j2++) {

  for(k2=0;k2<Nz;k2++) {
   S5: a = rho[i2][j2][k2];
   S6: tmp[k2] = a*c + b;
   S7: x[i2][j2][k2] = x[i2][j2][k2] + a*b; }

  for(k2=1;k2<Nz-1;k2++) {
   S8: z[i2][j2][k2] = x[i2][j2][k2] + tmp[k2+1] - tmp[k2-1]; }

} }
```

(a)

```
parfor(i=0;i<Nx;i++) {  // fused loop
 for(j=0;j<Ny;j++) {  // fused loop

  for(k1=0;k1<Nz;k1++) {
   a = rho[i][j][k1];
   tmp[k1] = a*b + c;
   x[i][j][k1] = x[i][j][k1] + a*c; }

  for(k1=1;k1<Nz-1;k1++) {
   z[i][j][k1] = x[i][j][k1] + tmp[k1+1] - tmp[k1-1]; }

  for(k2=0;k2<Nz;k2++) {
   a = rho[i][j][k2];
   tmp[k2] = a*c + b;
   x[i][j][k2] = x[i][j][k2] + a*b; }

  for(k2=1;k2<Nz-1;k2++) {
   z[i][j][k2] = x[i][j][k2] + tmp[k2+1] - tmp[k2-1]; }
} }
```

(b)

```
for(i1=0;i1<Nx;i1++) {
 for(j1=0;j1<Ny;j1++) {

  for(k1=0;k1<2;k1++) {
   a = rho[i1][j1][k1];
   tmp[k1] = a*b + c;
   x[i1][j1][k1] = x[i1][j1][k1] + a*c; }

  for(k1=2;k1<Nz;k1++) { // fused loop
   a = rho[i1][j1][k1];
   tmp[k1] = a*b + c;
   x[i1][j1][k1] = x[i1][j1][k1] + a*c;
   z[i1][j1][k1-1] = x[i1][j1][k1-1] + tmp[k1] - tmp[k1-2]; }

} }

for(i2=0;i2<Nx;i2++) {
 for(j2=0;j2<Ny;j2++) {

  for(k2=0;k2<2;k2++) {
   a = rho[i2][j2][k2];
   tmp[k2] = a*b + c;
   x[i2][j2][k2] = x[i2][j2][k2] + a*c; }

  for(k2=2;k2<Nz;k2++) {  // fused loop
   a = rho[i2][j2][k2];
   tmp[k2] = a*b + c;
   x[i2][j2][k2] = x[i2][j2][k2] + a*c;
   z[i2][j2][k2-1] = x[i2][j2][k2-1] + tmp[k2] - tmp[k2-2]; }

} }
```

(c)

Figure 5.1: (a) Original code, (b) Optimized (or transformed) code, and (c) Transformed program generated by PLuTo

-parallel'; O3 enables fusion) on original program, on an 8-core Intel Xeon processor. Performance results indicate the following,

1. Neither ICC (on the original program) nor PLuTo were able to either achieve loop fusion even though there is reuse, or mark the outermost loop as parallel for coarse-grained parallelization.

2. The transformed program in Figure 5.1b fuses both the loop nests, and thus achieves a sequential performance improvement of 1.19x over ICC. This demonstrates the efficacy of exploiting the data reuse across loop nests. PLuTo achieves fusion at the innermost loop within the same nest after some loop peeling and shifting. While this improves data reuse to some extent, it leads to non-vectorizable innermost loop due to the introduction of a forward dependence upon fusion. This explains PLuTo's worse performance as compared to ICC.

3. The transformed program in Figure 5.1b achieves a parallel performance improvement of 4.47x over ICC when running 8 threads in parallel, clearly revealing the optimization potential. Even after the two loop nests in Figure 5.1a are explicitly marked parallel (and variables a and tmp privatized), parallel performance improvement of the transformed program in Figure 5.1b is still 1.48x over ICC. This is because of the combined savings

of off-chip memory accesses and fork-join synchronization through loop fusion.

	1 thread	8 threads
Transformed code	1.19x	4.47x
PLuTo	0.48x	0.48x

Table 5.1: Performance speedup of transformed program in Fig 5.1b and PLuTo generated code (Fig 5.1c) wrt original code in Fig 5.1a

The key reason for the poor performance of ICC and PLuTo is that the use of same temporary variables in the two loop nests introduces dependences that are loop-carried on the outer loops of these nests. These dependences are not false since their removal will allow a perfect fusion of the two nests as far as the temporary variables are concerned, and this can potentially violate program correctness due to incorrect *definitions* reaching certain *uses*. The following section (Section 5.3), however, shows that these dependences as they exist are more stringent than needed, and artificially lead to an reduced degree of freedom for transformations such as fusion and parallelization. Further, Section 5.3 introduces concepts that lead us to *variable liberalization*, the technique that relaxes dependences to generate the transformed code as in Figure 5.1b.

5.2.1 Why not scalar and array expansion (perhaps, followed by contraction)?

Scalar or array expansion involves transforming the scalar or low-dimension array variables (such as a and $tmp[]$ in our motivating example) into full-dimension arrays (such as a[][][] and tmp[][][]). Expansion essentially creates a new memory location for the temporary variable for each iteration of the loop-nest. It thus removes loop-carried dependences between different references of the variable, leading to effective loop transformations. However, this technique (1) increases the memory footprint significantly and thus degrades temporal locality both in cache (74) and registers (75), and (2) requires declaration of new data structures and corresponding changes in the source code. The authors in (76; 77; 78) propose to perform array contraction to reduce the memory footprint after an initial expansion step. Their rationale is that expansion will enable aggressive optimizations, and array contraction will help to recover the loss in temporal locality at a later stage. But, the intermediate optimizations (such as loop distribution that can potentially distribute the definitions and their uses) hamper opportunities for such recovery through array contraction. The authors in (79; 74) attempt to control the expansion phase by constraining the aggressiveness of transformations so as to later effectively optimize the memory footprint. But, this often results in loss of optimization opportunities due to added constraints.

5.2.2 Why not scalar and array renaming across loop nests?

Renaming scalar and array variables in different loop nests could also result in disappearance of the transformation-limiting loop-carried dependences across loop nests, and will facilitate loop fusion and parallelization. But, there are three disadvantages of such a strategy which preclude us from implementing it.

1. The key limiting factor is that, after fusion of loop nests with (multiple) renamed temporary variables, the opportunity for reuse of data (accessed by temporary variables) in the higher levels of the memory hierarchy is lost. In addition, the working set in those higher (and smaller) levels of the memory hierarchy expands by a factor of the number of temporary variables and the number of merged nests. This degrades program performance especially in the presence of temporary array variables. For example, the transformed *zeusmp* benchmark application program from the SPEC Suite (containing 24 temporary arrays in a loop nest) runs 7% slower when loop fusion is performed after renaming, as compared to that performed without renaming (i.e. through *variable liberalization*). The performance degradation may be even larger for certain other scientific applications that use even more temporary variables.

2. As a result of this substantial increase in the number of temporary variables, the number of hardware prefetch streams also increase in the same proportion as each temporary array triggers one of those prefetch streams. Since every processor has a limited number of these streams, the transformed program can easily fall short of the needed prefetch streams, leading to performance degradation.

3. Another drawback is that renaming will require generation of new variable declarations and also modification of all references to the temporary variable in all loop nests.

Our framework, on the other hand, does not require any of those changes to the original source code, and more importantly, does not cause any unnecessary increase in the number of variables in the program. This promotes efficient data reuse in higher levels of cache and effective prefetching by the hardware.

5.3 Background

Instancewise dependence analysis (80) employed in state-of-the-art polyhedral compilers precisely tells which iterations of the involved statements are in a dependence, and is thus more effective is reasoning the feasibility of loop transformations. In this section, we use instancewise dependence analysis to show how dependences between temporary variables artificially suppress fusion of multiple loop nests. Since fusion and other supporting transformations (such as loop interchange and loop shifting) are simultaneously composed within the polyhedral framework, we consider them in conjunction when applicable. Therefore, we first show how loop interchange is hampered by dependences on temporary variables within the same loop nest.

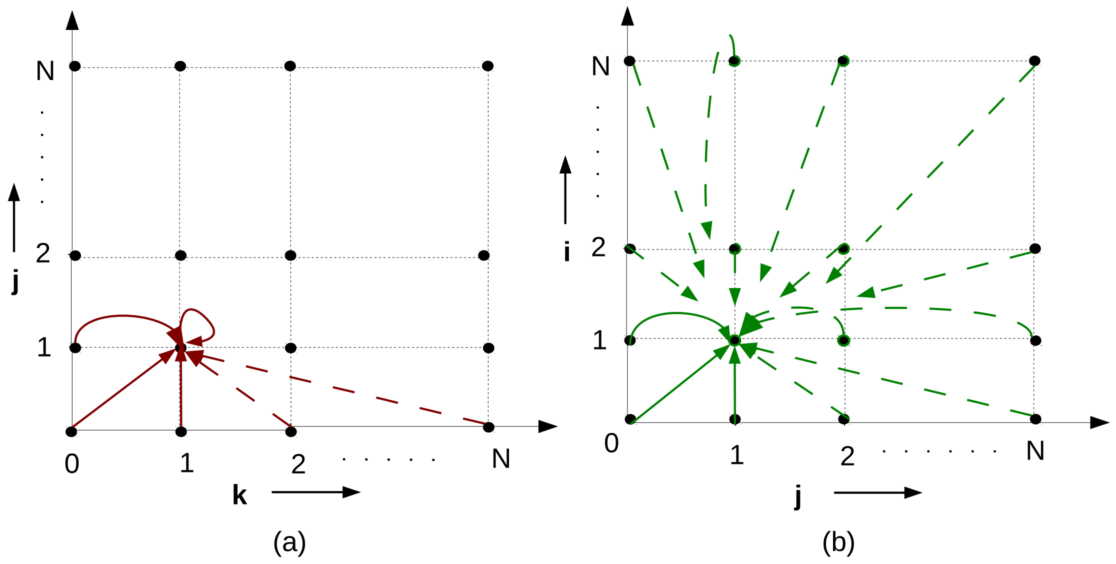


Figure 5.2: Instancewise (RAW) dependence between statements (a) S1 and S2, and (b) S2 and S8; the dashed arrows in the figure indicate a backward (RAW) dependence

Figure 5.2 shows the dependences between specific statement instances involving temporary variables to demonstrate the transformation limiting nature of such dependences. Since the same memory locations are accessed in multiple iterations of the loop nest, there are introduced loop-carried dependences in multiple loops of the loop nest. For example, consider the read to the temporary scalar variable a in Statement S2 in Figure 5.1a at the instance (j1=1, k1=1); it is the sink of a Read After Write (RAW) dependence from not just the instances (j1=1, k1=0) and (j1=1, k1=1) but also from instances (j1=0, k1=0 .. N-1) as shown in Figure 5.2a. Thus, it leads to loop-carried dependence in not just loop k but also loop j.

It is important to note that the dependence distance for the dependence involving the scalar a whose source instance is (j=0, 1<k<N), is negative along loop k (i.e. they are backward dependences), and such dependences are thus marked by dashed arrows in Figure 5.2 for emphasis. Since loop interchange requires all dependences to be either loop independent or forward-directed on the involved loops, interchanging loops j and k is rendered infeasible in this case in the presence of above-mentioned loop carried dependences. In this example, no loop-carried dependences on scalar a on the outermost loop i are shown because the scalar is privatizable at the outermost loop and thus, those dependences are ignored by the dependence analyzer to enable outer-loop or coarse-grained parallelization. This also enables loop interchange in loops i and j as after privatization, the dependences involving scalars become loop-independent in the outermost loop and are forward-directed in the loop j.

We next show how dependences on temporary variables in different loop nests precludes fusion of the involved loop nests. Figure 5.2b shows the instancewise dependence between statements S2 and S8 on the temporary array variable, tmp. Since S2 and S8 are in different nests, for a given $k = k1 = k2$, every write to $tmp[k1]$ in S2 will be visible to the read $tmp[k2]$ in S8 since the same memory location is involved each time. Thus, there is a RAW dependence from the definition in the first loop nest to the use in the second for every instance of loops i and j. This is depicted in Figure 5.2b for the instance (i=1, j=1) in the second loop nest, which becomes the sink for RAW dependences whose sources (0≤i≤N-1, 0≤j≤N-1) lie in the first loop nest. We call this an *all-to-all* dependence in loops i and j, since it exists from all iterations of these loops in the source-nest to all iterations of the corresponding loop in the destination-nest. Similarly, we can see that there is an all-to-all dependence in loop k involving the scalar a as shown in Figure 5.2a. Thus, if loop i or loop j were fused for both nests, then this would lead to a backward (negative distance) dependence on loop i or loop j. Such a fusion will be illegal, and the compiler restricts itself from performing it.
Similarly, the presence of temporary scalar variables in different nests leads to all-to-all dependences on all loops in the loop nest, which proves to be similarly fusion-restricting.

5.3.1 Some key concepts and definitions

We next describe some key concepts and definitions to aid the understanding of our approach to loop fusion by relaxing (or liberalizing) the dependences on temporary variables.

Live Ranges

In the context of polyhedral compilers, a live range is appropriately defined in terms of precise statement instances instead of static statements in the program as studied in earlier literature (81). We thus define a live range to be the range of statement instances from the definition instance of a value to its last use instance. For example, the live range for the scalar a defined in the first loop nest of the example program in Figure 5.1a is:

$$[\text{S1(i, j, k)} \rightarrow \text{S3(i, j, k)}], \text{ s.t. } 0 \leq \text{i} < \text{Nx}, 0 \leq \text{j} < \text{Ny}, 0 \leq \text{k} < \text{Nz}$$

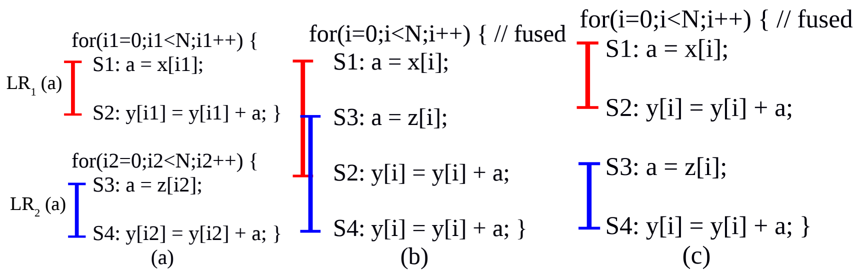
The above notation implies there is a live range that begins at the write in statement S1 in every iteration of the loop-nest (comprising of loops i, j and k) and lasts until statement S3 in the very same iteration.

Iteration-private Live Ranges

For the above live range, we note that the live range begins and ends in the same iteration of the innermost loop k of the loop nest. We call such a live range as *iteration-private* in loop k. Intuitively, the same live range is also iteration private in the outer loops, i and j. Past work (82) has shown that if all live ranges are iteration private in a loop, then that loop can be marked as privatizable. For example, the live range for the scalar a in our example program is iteration private in all 3 loops, and hence, the outermost loop is marked privatizable as noted in Section 4.1. The other loops are, however, not marked as privatizable because the focus there is just coarse-grained parallelization. We use this concept of iteration-private live ranges in our proposed *variable liberalization* optimization.

Live Ranges and Loop Fusion

As a result of loop fusion (after a possible relaxation of dependences on temporary variables), loop bodies of the fused nests merge. Consequently, multiple *definitions* and *uses* of temporaries with the same name end up in the same loop. Thus, to ensure legality of fusion, each use must see the same definition as in the original program, or in other words, *fusion of loop nests should preserve* ***non-interference of live ranges***. For example, the program in Figure 5.3a shows two live ranges for the scalar variable a in the two loop nests. Figure 5.3b shows a fused nest where the two live ranges interfere with each other as shown, and the first use of the scalar a doesn't see the same definition as in the original program; it is thus an incorrectly transformed program. Figure 5.3c preserves the non-interference of live ranges and is thus a correctly transformed program. Therefore, any relaxation of dependences must preserve this property to ensure correctness. *This (1) non-interference of live ranges, combined with (2) satisfaction of true (RAW) dependences form a criteria for legality of program transformation* as proved in previous work (83; 84). We use this criteria to reason correctness of the transformed program in the wake of our proposed relaxation of dependences involving (just) temporary variables.



```
      for(i1=0;i1<N;i1++) {
        S1: a = x[i1];
LR1 (a)
        S2: y[i1] = y[i1] + a; }

      for(i2=0;i2<N;i2++) {
        S3: a = z[i2];
LR2 (a)
        S4: y[i2] = y[i2] + a; }
                (a)
```

```
for(i=0;i<N;i++) { // fused
  S1: a = x[i];

  S3: a = z[i];

  S2: y[i] = y[i] + a;

  S4: y[i] = y[i] + a; }
              (b)
```

```
for(i=0;i<N;i++) { // fused
  S1: a = x[i];

  S2: y[i] = y[i] + a;

  S3: a = z[i];

  S4: y[i] = y[i] + a; }
              (c)
```

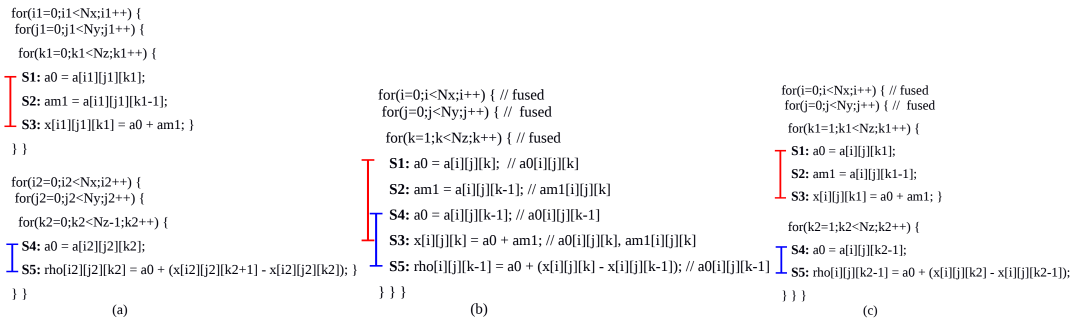
Figure 5.3: Live range interference

5.4 Our Approach

Our approach of achieving a legal fusion by relaxing the extra-stringent dependences on temporary variables is based on the following key insight:

5.4.1 Key Insight

The temporary variables by dint of their functionality of storing partial results temporarily in a program, are mostly defined in one of the inner loops of loop-nests and are used in the same



```
for(i1=0;i1<Nx;i1++) {
 for(j1=0;j1<Ny;j1++) {
  for(k1=0;k1<Nz;k1++) {
   S1: a0 = a[i1][j1][k1];
   S2: am1 = a[i1][j1][k1-1];
   S3: x[i1][j1][k1] = a0 + am1; }
} }

for(i2=0;i2<Nx;i2++) {
 for(j2=0;j2<Ny;j2++) {
  for(k2=0;k2<Nz-1;k2++) {
   S4: a0 = a[i2][j2][k2];
   S5: rho[i2][j2][k2] = a0 + (x[i2][j2][k2+1] - x[i2][j2][k2]); }
} }
              (a)
```

```
for(i=0;i<Nx;i++) { // fused
 for(j=0;j<Ny;j++) { //  fused
  for(k=1;k<Nz;k++) { // fused
   S1: a0 = a[i][j][k];  // a0[i][j][k]
   S2: am1 = a[i][j][k-1]; // am1[i][j][k]
   S4: a0 = a[i][j][k-1]; // a0[i][j][k-1]
   S3: x[i][j][k] = a0 + am1; // a0[i][j][k], am1[i][j][k]
   S5: rho[i][j][k-1] = a0 + (x[i][j][k] - x[i][j][k-1]); // a0[i][j][k-1]
} } }
              (b)
```

```
for(i=0;i<Nx;i++) { // fused
 for(j=0;j<Ny;j++) { //  fused
  for(k1=1;k1<Nz;k1++) {
   S1: a0 = a[i][j][k1];
   S2: am1 = a[i][j][k1-1];
   S3: x[i][j][k1] = a0 + am1; }

  for(k2=1;k2<Nz;k2++) {
   S4: a0 = a[i][j][k2-1];
   S5: rho[i][j][k2-1] = a0 + (x[i][j][k2] - x[i][j][k2-1]);
} } }
              (c)
```

Figure 5.4: (a) Example program, (b) (Incorrectly) Transformed program after dependence relaxation, and (c) Correctly transformed program

loop. In other words, they are *iteration private* in one of the inner loops, and consequently, in all outer loops. For example, in the example program in Figure 5.1a that has a 3-level loop nest, the temporary scalar a is iteration private in the innermost loop k. The temporary 1D array, tmp, is iteration-private in the next-to-innermost loop j. If there was a temporary 2D array, it would be iteration-private in the outermost loop i.

In the event of fusion of loop nests containing temporary variables with the same name, violation of both the criteria for legality with regards to temporary variables, i.e. (1) non-interference of live-ranges, and (2) satisfaction of RAW dependences, is only possible at loops within and including the innermost loop with iteration-private live ranges. Thus, *the dependences on the outer loops can be relaxed to allow fusion* of outer loops including the innermost loop with iteration-private live ranges. The 'relaxation' of dependences implies that the transformation-restricting all-to-all dependences on outer loops are converted into loop-independent dependences. This releases the necessary degrees of freedom for loop transformations. Thus, the RAW dependences on the temporary variables in the innermost loop with iteration-private live ranges are preserved just by the presence of a loop-independent dependence. Also, this relaxation is sufficient to preserve non-interference of live ranges and hence correctness of the transformed program subject to the following criterion:

Relaxation Criterion - *While dependences on temporary variables in different loop nests are relaxed, the resultant loop fusion should not be accomplished as a consequence of loop shifting in any of the fused loops.*

The rationale behind this *relaxation criterion* is that preservation of the live-range non-interference property cannot be guaranteed if loop shifting is performed as an enabling transformation for loop fusion. The following example clarifies the idea. Figure 5.4b shows a transformed program with fused nests, for the original program in Figure 5.4a. In this example, loop fusion is made possible after relaxing cross-nest all-to-all dependences on the scalar $a0$ to become loop-independent as discussed above. Thus, effectively, the scalar variables are expanded to become 3D arrays like other arrays such as x in the loop-nest. This is shown in the comments at the end of each statement.

In addition to loop fusion, the statements in the second nest are shifted by 1 iteration in the innermost loop (k) to prevent backward dependences on array x in loop k. In the transformed program, the (possible) backward dependence on array x is converted into a forward dependence through loop shifting to allow fusion. Thus, we see that there is a forward WAR dependence between statements S3 and S4 on the scalar $a0$ in loop k (i.e. $a0[i][j][k]$ written in statement S3 is read in the next iteration of the k-loop in statement S2) and a forward RAW dependence between statements S1 and S5 on the same scalar $a0$ in loop k (i.e. $a0[i][j][k]$ written in statement S1 is read in the next iteration of the k-loop in statement S5). As a result, the schedule of statements within the fused nest as shown in Figure 5.4b is completely legal from the point-of-view of data dependences (some of which have been purposely relaxed). However, the transformed program in Figure 5.4b is incorrect because the non-interference of live-ranges property is violated - the live ranges for the 2 def-use pairs involving the scalar $a0$ interfere with each other as shown. This happened precisely because the above-mentioned relaxation criterion is violated (due to shifting) in the innermost loop of the fused nest, loop k. The same is possible for the other loops in the nest, which justifies the constraint in its entirety.

In such cases when the relaxation criterion is violated for the last fused loop, the transformation for that loop is recomputed after enforcing an all-to-all dependence at that loop-level for the temporary variables in different loop nests. This causes loop distribution at that level to generate correct code as shown in Figure 5.4c. It is worth noting here that our relaxation (or legality) criterion requires knowledge of whether loop shifting was performed to enable fusion at each loop-level, to reason correctness of transformation in the wake of proposed dependence relaxation. The precise implementation of this in our framework is discussed in detail in Section 5.5.1.

It is also important to note that we use the criterion of non-interference of live ranges to reason about program correctness in the wake of relaxed data dependences on temporary variables. However, if the presence of certain other loop-carried data dependences on regular array references were to lead to a backward loop-carried dependence upon fusion of any two loops, then fusion of the concerned loops will not be performed as per the traditional criteria of performing fusion since such dependences are not modified (relaxed) by our framework.

Figure 5.4 demonstrated how the relaxation of dependences on outer loops would lead to a legal program transformation in the presence of temporary scalar variables in different loop nests that have the same loop order (provided the fusion does not violate our *relaxation criterion*). In the following subsection, we demonstrate how we similarly handle fusion of loop nests with temporary array variables. Then, we consider liberalization of dependences when the involved nests have different loop order and show that loop-order determines depth of fusion of loop nests.

5.4.2 Fusion of loop nests with same loop-order in the presence of temporary array variables

We use our original example program (Figure 5.1a) to illustrate *liberalization* in the presence of temporary array variables. As discussed in Section 5.3, fusion of the 2 loop nests in Figure 5.1a is prevented by all-to-all dependences on the temporary 1D array, tmp, in all but the innermost loop in the nest. Similar to the case of scalar variables, we allow for loop fusion in this case by relaxing the all-to-all dependences to become loop-independent in all loops (i.e. loops i and j) including the innermost loop with iteration-private live ranges. In order to preserve live-range non-interference in loop j, we force an all-to-all dependence on the inner loop k that ensures that the two loop bodies will be distributed at the innermost loop-level as shown in Figure 5.1b. Thus, the net effect is that the WAR dependence between S2 and S8 and the RAW dependence between S4 and S6 is relaxed from being all-to-all in loops i and j to just being all-to-all in the innermost loop k, or in other words, the dependence has been shifted to the innermost loop where absolutely needed. Since no shifting was performed to enable loop fusion, the transformed program under the relaxed dependences is correct.

5.4.3 Fusion of loop nests with different loop-orders

```
for(i1=0;i1<Nx;i1++) {
 for(j1=0;j1<Ny;j1++) {

  for(k1=0;k1<Nz;k1++) {
   tmp[k1] = a*b + c; }

  for(k1=1;k1<Nz-1;k1++) {
   x[i1][j1][k1] = tmp[k1+1] - tmp[k1-1]; }

} }

for(j2=0;j2<Ny;j2++) {
 for(i2=0;i2<Nx;i2++) {

  for(k2=0;k2<Nz;k2++) {
   tmp[k2] = a*c + b; }

  for(k2=1;k2<Nz-1;k2++) {
   y[i2][j2][k2] = tmp[k2+1] - tmp[k2-1]; }

} }
```
(a)

```
for(i=0;i<Nx;i++) { // fused loop
 for(j=0;j<Ny;j++) { // fused loop

  for(k1=0;k1<Nz;k1++) {
   tmp[k1] = a*b + c; }

  for(k1=1;k1<Nz-1;k1++) {
   x[i][j][k1] = tmp[k1+1] - tmp[k1-1]; }

  for(k2=0;k2<Nz;k2++) {
   tmp[k2] = a*c + b; }

  for(k2=1;k2<Nz-1;k2++) {
   y[i][j][k2] = tmp[k2+1] - tmp[k2-1]; }
} }
```
(b)

Figure 5.5: (a) Loop nests with different loop-orders, (b) Loop fusion after interchange and liberalization

In an application program, loop nests may contain different loop-orders as shown in Figure 5.5a. Loop fusion in such cases is more involved. However, there may still be an opportunity for partially fusing the candidate loop nests, either because of common outer loops or through loop interchange in one of the nests. Thus, considering loop fusion in conjunction with loop interchange becomes particularly relevant.

In case of temporary 1D arrays, loop nests are imperfectly fused at the innermost loop level as shown in Figure 5.5a, and thus interchanging the innermost two loops in infeasible. However, the outer loops can still be interchanged after array privatization in the outermost loop, which is similar to the way scalar privatization allows interchange as discussed in Section 5.3. In case of scalars, we extend scalar privatization to the inner loops also in order to allow for loop interchange in all loops of the loop-nest. Once loop interchange is made feasible, the process of relaxation of dependences on temporary variables in different nests is similar - the all-to-all dependences on all outer loops upto and including the innermost loop with iteration-private live ranges are relaxed, and an all-to-all dependence on remaining inner loops, if any, is introduced to preserve the non-interference of live-ranges property. For temporary 1D arrays as in our example program in Figure 5.5a, the all-to-all dependences on loops i and j in the first nest to loops j and i, respectively, in the second nest, are relaxed, and an all-to-all dependence is introduced in the inner loop k to achieve the transformed program in Figure 5.5b. It is important to note that fusion of all loops with iteration-private live ranges was possible since such loops were common (i.e. loops i and j in the first nest and loops j and i in the second) in this case.

When trying all possible combinations of loop-orders in the two nests, we find that we are able to fuse the nests at least up to the depth of 1 loop-level in all cases, since the two loops with iteration-private live ranges will always have at least 1 loop in common, and loop interchange ensures that the common loop can be the outermost loop in the fused nest. Loop fusion even at the outermost loop, ensures data reuse in the last level cache in most cases, and is thus valuable in various scientific benchmarks where different loop orders in loop nests is common.

5.4.4 Cases in which dependence relaxation is not feasible

In this section, we mention two examples where dependences on temporary variables in different loop nests cannot be relaxed because the conditions for such relaxation are not satisfied. Figure 5.6a shows the first example that contains a temporary array tmp in the two loop nests. Relaxing dependences on the outer loops i and j would result in fusion of the two nests, but

```
for(i1=0;i1<Nx;i1++) {
 for(j1=0;j1<Ny;j1++) {
  for(k1=0;k1<Nz;k1++) {
   tmp[k1] = tmp[k1] + a[i1][j1][k1]; }
} }

for(i2=0;i2<Nx;i2++) {
 for(j2=0;j2<Ny;j2++) {
  for(k2=0;k2<Nz;k2++) {
   tmp[k2] = tmp[k2] + b[i2][j2][k2]; }
} }
```
(a)

```
for(i=0;i<Nx;i++) {
 for(j=0;j<Ny;j++) {
  a[i] += b[i][j] + c[i][j];
} }
```
(b)

Figure 5.6: Example programs where dependence relaxation is infeasible

such a fusion will clearly be illegal and our framework desists from performing it. The reason is that the live range involving tmp is not iteration private in the outer loops of the loop-nest, i.e. a value written in $tmp[k]$ in each iteration of loop j is read in the next iteration of loop j. The same holds true for the outermost loop i as well. The net result is that tmp is live-in in the second nest, which clearly indicates the infeasibility of loop fusion in this case.

Figure 5.6b shows the second example. The subscript of the temporary array a is the outermost loop variable i as opposed to the previous examples where temporary array's subscript is an innermost loop variable. Clearly, the live-range of a is not iteration-private in any loop in the loop-nest, which also indicates its use later in the program. It is therefore not marked for *liberalization*. Also, in such a case, *liberalization* is not needed to allow fusion (and parallelism) of the outer loop since there is no all-to-all dependence on the outer loop.

5.5 Implementation: putting it all together

This section describes the algorithm used to implement *variable liberalization*, as discussed in the preceding sections. The algorithm comprises two steps. Step 1 essentially performs variable privatization in all outer loops of a loop-nest that have iteration-private live-ranges for all references in the loop body. We perform privatization in multiple outer loops instead of just the outermost loop. It allows loop interchange which creates opportunities for loop fusion. This is effected by removal of the all-to-all dependences (identified by Line 6 in the algorithm) on temporary variables in the iteration-private loops of the loop-nest containing the source statement (Line 5). Step 1 is performed only for those dependences whose sink (or destination) statement is live for its source statement (Line 4). This condition holds for temporary variables being defined and used in the same loop-nest, and thus program correctness can be preserved by only retaining a loop-independent dependence between the source and sink in the innermost loop with iteration-private live ranges after pruning the all-to-all dependences.

ALGORITHM 2: Variable privatization

```
1:  INPUT:
    $deps$ : Copy of the list of dependences
2:  STEP 1:
3:  begin
4:  for each dependence $d \in deps$ s.t. IsLive($d$.src_stmt,$d$.dest_stmt) = $true$ do
5:     for each loop $l \in$ LoopNest($d$.src_stmt) s.t. IsIterPriv($l$,$d$.src_stmt) = $true$ do
6:        if IsAllToAll($d$,$l$) then
7:           Remove $d$ from the list of dependences, $deps$
8:  end
9:  STEP 2:
10: begin
11: for each dependence $d \in deps$ s.t. IsLive($d$.src_stmt,$d$.dest_stmt) = $false$ do
12:    for each loop $l \in$ LoopNest($d$.src_stmt) do
13:       if IsIterPriv($l$,$d$.src_stmt) = $true$ then
14:          Make $d$ loop-independent at loop-level $l$
15:       else
16:          Make $d$ all-to-all at loop-level $l$
17: end
18: OUTPUT: Relaxed set of dependences, deps
```

It is important to note here that the condition in Line 4 does not hold for those dependences whose source and sink lie in different loop nests, or even when they lie in the same loop nest in case of multiple definitions of the temporary variable in the same nest. It is such dependences that restrict loop fusion and require more careful treatment. Therefore they are handled separately in Step 2 to preserve the non-interference of live-ranges property. This involves forcing an all-to-all dependence on the loops inner to the innermost iteration-private loop (Line 16) that strictly precludes fusion of the two nests beyond the innermost iteration-private loop, whereas dependences on the other outer loops are relaxed to become loop-independent (Line 14). This ensures program correctness, provided the relaxation criterion is not violated.

5.5.1 Validation of relaxation criteria

In polyhedral compilation, violated dependence analysis (80) is becoming increasingly popular. Using violated dependence analysis, the polyhedral compiler can reason the correctness of the proposed transformation in the wake of either relaxed legality checks (85) or relaxed dependences (84; 86). In such an approach, one has to wait until a transformation is performed and re-iterate the process possibly several times in the case of an incorrect transformation, or take a corrective action. In this work, we introduce *violated transformation analysis*, where we can reason about the correctness of the computed transformation during the process of finding transformation itself, and the recovery is executed immediately to yield a correct (although weaker) transformation.

As discussed in Section 5.4, we can reason about the validity of dependence relaxation using our *relaxation criterion* which amounts to determining the absence of loop shifting transformation in the fused loops. In our framework, we determine this for each step of the transformation process that involves determining what statements can be fused in the same nest at a given loop-level, starting from the outermost loop[2]. If the transformation computed for the statements from different nests indicates that loop shifting has occurred in the last (fused) loop, then the last loop found is discarded. In such a case, an all-to-all dependence is introduced at that loop to ensure that the 2 loop nests are distributed at that loop-level to guarantee correctness. Thus, the correctness is guaranteed during the transformation process itself and there is no need to re-iterate the process.

For example, for the program in Figure 5.6a, when trying all possible combinations of loop orders in the two nests, we find that for certain combinations such as (i-j-k) and (k-i-j), the framework initially finds two common loops and the common loop i with iteration-private live ranges becomes the outermost loop. However, for the second loop found, there is loop shifting in S4 by 1 with respect to other statements, and thus the second loop is discarded. The final transformed program contains just 1-deep fused nest. Thus, we find violated transformation analysis to be extremely useful in this case, and we believe that this technique can become useful to find or reason about correctness of other transformations as well.

[2]More detail on the process of computing transformations using a polyhedral compiler can be found in (6)

5.6 Experimental Evaluation

Fusion Model	Description
gfortran	GNU Fortran Compiler (baseline); flags = ‘-O3 -ftree-loop-parallelize=n’
ifort	The Intel Fortran Compiler; flags = ‘-O3 -parallel’
smartfuse	The default fusion model in PLuTo. It uses heuristics to determine a good fusion schedule
explicit	Explicit parallelization or parallelization by hand
var-lib	Variable liberation in PLuTo (our work)

Table 5.2: Summary of fusion models in different compilers

5.6.1 Setup

The test programs were compiled and run on an Intel Xeon processor (E5-2650) with 8 Sandy Bridge-EP cores, operating at 2.0GHz. The processor has private L1 (32KB per core) and L2 (256KB per core) caches and a 20MB shared L3 cache, and 16GB memory. Since we implement the *variable liberalization* optimization in the PLuTo polyhedral compiler, we compare our performance results with the fusion model within PLuTo, in addition to the popular production compilers, the GNU and the Intel compilers. A summary of the different fusion models used for comparison is given in Table 5.2. It is important to note that PLuTo itself cannot parse Fortan code (and the applications we use for experiments are written in Fortran). We thus use PolyOpt/Fortran (87), a tool that uses ROSE compiler (88) frontend to parse Fortran code and relies on PLuTo (version 0.5.4) for loop optimizations. The transformed source code generated is then compiled using the Intel compiler v14 (ifort) as the backend compiler. The compile time options used with the Intel compiler are ‘-O3’ and ‘-parallel’.

5.6.2 Benchmarks

The benchmarks used in the experiments include 4 real application programs used in the scientific community, and in other published research. A brief description of these benchmarks is given in Table 5.3. In these 4 applications, we identify 8 hot regions spanning 5 subroutines. Each identified hot region constitutes a Static Control Part (SCoP), or the maximal syntactic program segment that contains sequences of loop nests with constant strides and affine bounds. As a result, all chosen hot regions are amenable for optimizations by a polyhedral compiler. It is

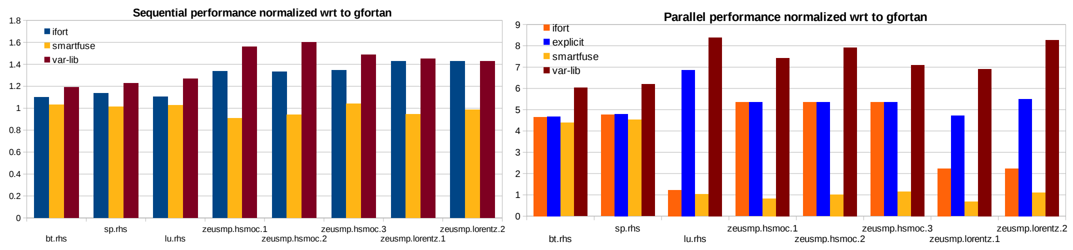


Figure 5.7: Performance results using (a) single thread (b) 8 threads (cores)

important to note that each of these 8 SCoPs contain multiple large loop-nests with the number of statements in each SCoP ranging from 48 to 121. Such large sequences of statements are known to be hard for the compilers to optimize.

Benchmark	Benchmark Suite	Category	Problem Size
applu	NPB/OMP2012	Computational Fluid Dynamics (CFD)	N=102; CLASS B
bt	NPB/OMP2012	”	”
sp	NPB	”	”
zeusmp	CPU2006	Simulation of astrophysical phenomena	Reference Input

Table 5.3: Summary of the benchmarks

5.6.3 Results and Discussion

Figure 5.7 shows the performance results for the 8 SCoPs using different compilers for comparison. Figure 5.7a shows results for sequential performance, while 5.7b shows parallel performance. Among the 4 benchmarks, the entire *rhs* subroutine in *bt, sp and lu* benchmarks forms a single SCoP, the *hsmoc* subroutine within *zeusmp* benchmark consists of 3 SCoPs each separated by a procedure call (with possible side-effects and thus recognized as a non-affine component) whereas the *lorentz* subroutine consists of a single SCoP, but divided into 2 in order to limit the memory requirement for optimizing it using polyhedral compilation. Each of the chosen benchmarks contains multiple loop nests and thus offers considerable opportunities for loop optimizations - a characteristic of most scientific application codes.

In particular, each of the chosen benchmarks contain loop-nests with different loop-orders, and thus are more challenging from the point of view of the loop fusion optimization. From the figure, we can find that *var-lib*, i.e. the compiler optimization proposed in this work outperforms the other compilers in almost all cases. The sequential performance of *var-lib* outperforms *ifort* by as much as 1.2x, and the performance improvement is significantly larger for parallel versions of the benchmarks with *var-lib* outperforming *ifort* by as much as 6.8x in *lu*. Even against the explicitly parallelized versions, *var-lib* performs considerably better as shown in Figure 5.7b. We next discuss the performance results for each compiler in more detail.

gfortran and ifort

gfortran or the GNU Fortran compiler was chosen as the baseline in our experiments. In addition, we also show results with the Intel Fortan compiler. In all cases, *gfortran* proved to be worse than *ifort* due to the latter being much more effective at vectorization. However, both these production compilers are equally poor in performing loop fusion. As a result, neither of them fused any large loop-nests for any of the SCoPs listed in the figure. *gfortran* performed the worst of all compilers because it could not recognize parallel loops in any benchmark. *ifort* could recognize parallel loops in *bt.rhs*, *sp.rhs*, *zeusmp.hsmoc*, and in *zeusmp.lorentz.1*.

From these results (and other experiments using our test kernels), we find that *ifort* is capable of parallelizing the loop-nest in the presence of temporary scalar variables, but not in the presence of temporary array variables. It is for this reason that *ifort* could not parallelize *lu*. Although, temporary array variables exist in *hsmoc* as well, we believe that *ifort* relies on recognizing certain specific patterns in this case to achieve parallelization because the inability to recognize temporary array variables in a less computationally intensive subroutine, *lorentz*, in the same benchmark hurts parallelization opportunity. In any case, we can conclude that existing production compilers are limited in their capability of detecting parallelism in scientific application codes that contain temporary variables, and much more so in performing the important loop fusion optimization.

PLuTo's smartfuse

PLuTo is a state-of-the-art polyhedral compiler that has shown significant promise in achieving automatic loop parallelization (89). PLuTo uses three different heuristics for fusion, *min-, max- and smartfuse*. *maxfuse* and *smartfuse* are practically equivalent for SCoPs with many statements, and *minfuse* performs maximal distribution and almost always performs sub-optimally. Thus we only choose to compare with *smartfuse*. PLuTo is also capable of performing scalar privatization that empowers it to perform coarse-grained parallelization in the presence of scalar temporary variables. It cannot, however, privatize temporary array variables. As a results, *smartfuse* can identify parallel loops in *bt* and *sp*, but not in any of the other SCoPs. Also, since

PLuTo does not relax dependence on such temporary variables across loop-nests or, in other words, perform *variable liberalization*, it is deprived of the opportunity to perform loop fusion. This amounts to reduced performance as compared to *var-lib* even for the benchmarks where it could achieve parallelization.

Variable Liberalization (var-lib)

When using a single thread to run benchmarks, *var-lib* outperforms all other compilers on account of fusing multiple loop-nests. The improvement is proportional to the fusion opportunity available in the benchmarks. For example, both the large nests in *zeusmp.hsmoc.2* are fused to enable data reuse, whereas in the other cases such as *lu*, only 2 of the 3 large nests could be fused together. This is because the common outer loop of the first two nests in *lu* is the innermost loop of the third nest, and the innermost loop cannot participate in loop interchange because of the presence of temporary arrays and imperfect nesting.

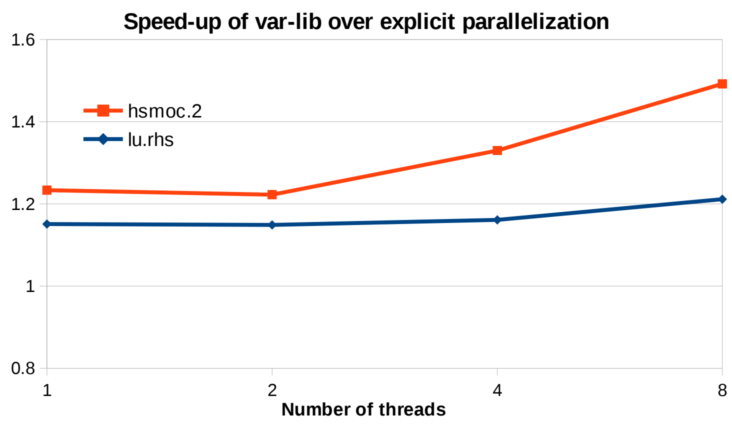


Figure 5.8: Parallel performance comparison of *var-lib* and *explicit parallelization for* rhs *subroutine in* lu

Interestingly, the performance improvement for all benchmarks surges upon parallelization even for benchmarks that are successfully parallelized by other compilers (including explicitly or manually parallelized code) such as *bt, sp and zeusmp.hsmoc*. This is due to two reasons - (1) reduction of fork-join synchronization points, and more importantly, (2) saving of off-chip memory accesses and thus the bandwidth, which is a source of contention among parallel threads in such memory-intensive applications. Both of these benefits are direct consequences of effective loop fusion achieved as a result of *variable privatization*. In addition, *var-lib* significantly outperforms all other compilers when they cannot identify outer-parallel loops due to the presence of temporary array variables and imperfect nests as in *lu* and *zeusmp.lorentz*.

Hardware Event	$\frac{\text{explicit-varLib}}{\text{explicit}} * 100\ \%$	
	lu	zeusmp
Load_latency_gt_512	24.03	12.88
Resource_stalls.any	7.68	13.8
Uops_dispatched.stall_cycles	8.8	19.94

Table 5.4: Performance counters indicating reduced pipeline stalls and effective memory latency through *var-lib*

Figure 5.8 shows the speed-ups achieved by *var-lib* over *explicitly* parallelized code for two of the eight SCoPs, *lu.rhs* and *zeusmp.hsmoc.2* when using different number of threads. The speed-up achieved in case of *zeusmp.hsmoc.2* is larger than that in *lu.rhs*. This is because of a greater opportunity of fusion in the former case, as explained earlier. Further, the speed-up achieved increases with the increase in the number of threads. This is explained with the help of performance counters shown in Table 5.4 as obtained from Intel's VTune Performance Profiler (53). Both benchmarks witness a considerable reduction (24.03% and 12.88%, respectively) in the number of memory loads whose latency is greater than 512 cycles after *variable liberalization*. Since the memory latency on the test processor (SandyBridge microarchitecture) is roughly 200 cycles, such high latency corroborates bandwidth contention. Thus, clearly, lower number of such high latency loads confirms the efficacy of effective loop fusion performed by *var-lib*. These high latency loads result in increase in the number of stall cycles due to resource

Benchmark	Subroutine	# statements	# deps	% Ex. time	Compile time (s)/Memory req. (MB)			$S = \frac{\text{Ex. Time}_{\text{ifort}}}{\text{Ex. Time}_{\text{var-lib}}}$
					ifort	smartfuse	var-lib	
bt	rhs	48	1419	16.8	4/116	303/703	180/281	1.06x
sp	rhs	50	1428	33.7	4/116	360/766	213/301	1.08x
lu	rhs	106	3033	63.1	1.9/71.7	11302/5542	3247/1551	2.17x
zeusmp	hsmoc.1	121	2660	24.1	5/141	10700/10780	162/4060	1.36x
	hsmoc.2	118	2493		5.9/151	9067/9693	246/6361	
	hsmoc.3	120	2296		5.7/143	7739/9014	280/5759	
	lorentz.1	98	2227	27.3	3.1/95.2	6027.5/10053	165/3222	
	lorentz.2	92	2149		2.49/86	5833/8275	1114/5152	

Table 5.5: Compile time, memory requirement of different compilers and overall (application) speedup (S) achieved by *var-lib* over *ifort*

contention (measured by the counter *Resource_stalls.any* that includes stalls due to fully occupied load Buffer, Store Buffer, Reorder Buffer, and Reservation Stations) and also due to reservation stations waiting on operands (measured by the counter *Uops_dispatched.stall_cycles*). The reduction in loads with high latency are larger for *lu* than *zeusmp* since the optimized subroutine, *rhs*, in *lu* is its most memory-intensive part and thus benefits more from loop fusion. However, reduction in stalls is more significant in case of *zeusmp* since the optimized subroutines, *hsmoc* and *lorentz* together contribute more than 50% of the execution time in *zeusmp*, as compared to *rhs* that contributes only 20% of the total execution time in the explicitly parallelized version of the *lu* benchmark.

Table 5.5 compares the compile time and memory requirement of the different compilers in each category. For example, *ifort* is the representative of the existing production compilers, *smartfuse* is the representative of existing polyhedral compilers, and *var-lib* is our work. Clearly, the time and memory needed by *ifort* for compilation is much less than any of the polyhedral compilers. This is because, existing production compilers consider a very limited scope for analysis such as a single loop-nest or consecutive loop nests that have the same bounds and same loop order. In other words, they trade performance for the time and memory spent in analysis to reason transformations.

The polyhedral compilers, on the other hand, consider all possible dependences in a given SCoP to reason transformations. They thus trade time and memory for performance. In some cases, such as for *lu* benchmark, the compile time (and memory requirement) can be extremely large because of their multiple imperfectly nested loops. Since *variable liberalization* relies on pruning dependences within a loop-nest (to enable loop interchange) and relaxing dependences across loop-nests, it is also effective in significantly reducing both the compile time and memory requirement, while achieving superior transformed code with effective fusion and parallelization. On average, *var-lib* achieves a reduction of 22x and 2.4x in terms of compile time and memory requirement, respectively. Lastly, Table 5.5 also shows that *var-lib* achieves an overall (application) performance improvement ranging from 1.06x in *bt* where the optimized subroutine *rhs* only contributes 16.8% to the overall execution time of the benchmark, to 2.17x in *lu* where the optimized subroutine contributes 63.1% to the overall execution time.

5.7 Related Work

Past work has well recognized the importance of enabling important optimizations through variable privatization and expansion. Variable privatization (90; 82; 91) involves creating multiple copies of a temporary variable (scalar or a low dimension array) that is defined and used in one of inner loops of a loop nest (i.e. has iteration-private live-ranges in that loop). Consequently, each thread is assigned a private copy of the variable, thus eliminating races between threads and allowing coarse-grained parallelism at an outer-loop. From a data dependence analysis perspective, privatization involves pruning loop-carried dependences on the temporary variables, at the outermost loop. However, privatization only affects a single loop-nest; the data dependences between temporary variables in different nests continue to be stringent and thus, transformation-restricting.

While privatization is a supporting optimization for parallelism, expansion (92) is another enabling optimization for many other transformations such as loop fusion, shifting and distribution. However, it has a known drawback of significantly increasing the memory footprint. To counteract this, the authors in (76; 77; 78) propose to perform a maximal expansion to enable important optimizations, and then attempt to contract the expanded variables as much as possible. However, this contraction of the optimized program is not only difficult, but is sometimes not possible in the wake of transformations such as distribution that can potentially distribute the definition and use of expanded variables to different loops. In our work, we propose variable liberalization that is a combination of privatization and expansion in that it achieves the benefits of both. In liberalization, we relax dependences on temporary variables in different nests as in privatization, thus effectively expanding those variables, but not through an actual expansion in their dimensions.

In the past, there has also been considerable work on loop fusion using different algorithms to find the best loops to fuse with the objective of maximizing data reuse (72; 71), minimizing synchronization (71), reducing register pressure (73), and saving off-chip bandwidth (65). Interestingly, none of these techniques have been used in existing production compilers for various pragmatic reasons in addition to the increased compile time. These pragmatic reasons include loops with different bounds, orders, or those that are imperfectly nested and are hard for the compiler to optimize in real application codes. However, we show in this work that even if these limitations were removed (most of which are non-existent in polyhedral compilers due to exact dependence analysis), effective fusion and consequent parallelization could still be prohibited due to the occurrence of transformation-restricting dependences on temporary variables in different nests. Thus, this work proposes *variable liberalization* to fill this gap.

Recently, the authors in (80) have proposed violated dependence analysis which has made it possible to reason the correctness of an applied transformation by polyhedral compilers even when such a transformation is built upon a relaxed set of dependences. Vasilache et al. (85) propose a framework for correcting violations to a desired transformation through other enabling transformations such as loop shifting, index-set splitting and code motion. However, it is unclear that the framework will always successfully correct the transformation performed. In the event of a failure, a different (new) transformation is applied and correction of it is attempted iteratively, resulting in possibly multiple passes.

The authors in (84; 86) propose lazy expansion where they ignore WAW and WAR dependences when finding transformations and then reason correctness using an extension to the violated dependence analysis, called live range violation analysis. In the event of a violation, variables are expanded as needed (lazily) to ensure correctness. Since they do not violate true (RAW) dependences, they guarantee transformation correctness, albeit at the cost of higher memory footprint. However, these works do not focus on loop fusion, which is crucial for performance in large SCoPs. In fact, since there also exist RAW dependences across loop-nests which are fusion-restricting, considering them for transformations without relaxation will disallow fusion in the first place in the presence of temporary variables in different nests. It is also important to note that even partial expansion and renaming can significantly (potentially double with the fusion of just two nests) the working set in the higher levels of the memory hierarchy by increasing the amount of data accessed in the inner loops of the nest, and thus hurt performance. In this work, we extend the violated dependence analysis to our proposed violated transformation analysis where we can reason if the transformation has gone wrong during the time it is being found, and immediately take corrective measures by imposing stricter constraints. This thus prevents subsequent passes to find a correct transformation, and does not require any renaming or expansion.

5.8 Conclusion

In this work, we propose *variable liberalization*, a compiler technique that strategically relaxes dependences on temporary variables that impede useful optimizations such as fusion of loop-nests. Unlike *variable expansion*, *variable liberalization* does not cause an actual expansion of variables while enabling fusion, thus further improving the memory performance of transformed programs. In our framework, we guarantee correctness through *violated transformation analysis*, a technique that validates the relaxation of dependences for the purpose of the fusion optimization without requiring an iterative process in achieving a correct transformation. Experimental results demonstrate its effectiveness to achieve fusion of nests (and subsequent parallelization of the fused nest). It can thus substantially increase data reuse in real application programs leading to improved performance.

Chapter 6

Loop Fusion and Real Applications - Part 2

6.1 Introduction and Motivation

In the last chapter, we identified 2 key challenges faced by the compiler when attempting to perform global program transformations. Also, we presented a solution to meet the first challenge, namely, existence of transformation-limiting dependences on temporary variables across loop-nests. In this chapter, we address the second important challenge, i.e. deciding on a cost model to find achieve effective fusion structures that achieve best performance on current processors. This complements the work presented in the last chapter, so as to present a complete solution to enable effective global program transformations by the compiler.

Since we implement our solution to global program transformation in a polyhedral compiler, we first discuss some of the limitations in traditonal compilers that are overcome by polyhedral compilers. This justifies our choice of the latter class of compilers when attempting global program transformations. Past work (93; 94; 95) on loop fusion using traditional compilers did not consider loop fusion in the context of other transformations. This misses some of the better solutions in various cases. For example, in the *gemver* benchmark shown in Figure 6.1(a), traditional compilers first identify outer parallel loops and then consider loops for fusion such that outer-loop or coarse-grained parallelism is not hurt. This approach cannot achieve fusion of statements S1 and S2 because of unsatisfied dependences (as shown in Figure 6.1(b), and

```
for (i=0; i<N; i++)  // parallel loop
 for (j=0; j<N; j++)
  S1: B[i][j] = A[i][j] + u1[i]*v1[j]
                         + u2[i]*v2[j];
for (i=0; i<N; i++)  // parallel loop
 for (j=0; j<N; j++)
  S2: x[i] = x[i] + beta*B[j][i]*y[j];
for (i=0; i<N; i++) // parallel loop
 S3: x[i] = x[i] + z[i];
for (i=0; i<N; i++) // parallel loop
 for (j=0; j<N; j++)
  S4: w[i] = w[i] + alpha*B[i][j]*x[j];
```

(a)

```
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
  S1:  B[i][j] = A[i][j] + u1[i]*v1[j] + u2[i]*v2[j];

  S2:  x[i] = x[i] + beta* B[j][i]*y[j];
 . . .
```

(b)

```
for (i=0; i<N; i++)   // parallel loop
 for (j=0; j<N; j++)
  S1:  B[j][i] = A[j][i] + u1[j]*v1[i] + u2[j]*v2[i];
  S2:  x[i] = x[i] + beta* B[j][i]*y[j];
 . . .
```

(c)

Figure 6.1: (a) the *gemver* kernel; (b) illegal fusion; (c) legal fusion after loop interchange in the first loop-nest;

thus data reuse opportunity is lost. However, if loop fusion was considered in a manner coupled with outer-loop parallelism and loop interchange, legal fusion could be accomplished after interchanging loops i and j in the first loop nest as shown in Figure 6.1(c). As a result, the transformed program now achieves both data reuse and outer-level parallelism.

In addition, in the traditional compilers, the granularity of loop fusion is an entire loop nest. This again misses good solutions. For example, in the *swim* benchmark shown in Figure 6.2, the two loop nests cannot be entirely fused because of dependences between some of the intermediate statements (S4-S12) and statements S13, S14 of the second loop nest. However, if the granularity of loop fusion was individual statements, statements S15 and S18 that are not dependent on any of the intermediate statements could be fused with statements S1-S3 in the first loop nest, as shown later in Figure 6.4(b). In addition, the state-of-the-art production compilers only consider consecutive loops for fusion in a pair-wise fusion (96), which clearly limits their capability of achieving global reuse in the program.

The polyhedral compiler framework is markedly different from its traditional counterparts in that, it takes a *statement centric view* of the entire program as against a *loop-nest centric view*. This shift in the intermediate representation allows to simultaneously compose multiple high-level loop transformations and also to reduce the granularity of fusion from loop nests to individual statements. This enables finding effective fusion partitions that exploit data reuse in a global program context, those that are missed by the traditional compilers as shown in the above two examples. *However, this statement centric view also results in a very large search space of the possible fusion partitionings.* The size of the search space varies exponentially with the

```
for (i=0; i<M; i++) {
 for (j=0; j<N; j++) {
   S1: UNEW[i+1][j] = UOLD[i+1][j]+C1*(Z[i+1][j+1]+Z[i+1][j])
        *(CV[i+1][j+1]+CV[i][j+1]+CV[i][j]+CV[i+1][j])-C2*(H[i+1][j]-H[i][j]);
   S2: VNEW[i][j+1] = VOLD[i][j+1]-C1*(Z[i+1][j+1]+Z[i][j+1])
        *(CU[i+1][j+1]+CU[i][j+1]+CU[i][j]+CU[i+1][j])-C3*(H[i][j+1]-H[i][j]);
   S3: PNEW[i][j] = POLD[i][j]-C2*(CU[i+1][j]-CU[i][j])
        -C3*(CV[i][j+1]-CV[i][j]);
  }
}
/*   Update of variables at grid boundary   (S4 – S12)    */
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
    S13: UOLD[i][j] = U[i][j]+ALPHA*(UNEW[i][j]-2*U[i][j]+UOLD[i][j]);
    S14: VOLD[i][j] = V[i][j]+ALPHA*(VNEW[i][j]-2*V[i][j]+VOLD[i][j]);
    S15: POLD[i][j] = P[i][j]+ALPHA*(PNEW[i][j]-2*P[i][j]+POLD[i][j]);
    S16: U[i][j] = UNEW[i][j];
    S17: V[i][j] = VNEW[i][j];
    S18: P[i][j] = PNEW[i][j];
   }
 }
/*   Update of variables at grid boundary   (S19 – S36)   */
```

Figure 6.2: the *swim* benchmark

number of statements, or more precisely, with the number of Strongly Connected Components (SCCs). For example, for just the 3 statements S1-S3 in the first loop nest of *swim* that have no dependences between them, there can be 3! (=6) different ways in which they can be ordered. Further, for a particular ordering of statements (e.g., S2-S3-S1), there can be 2^{3-1} (=4) different partitionings[1], i.e. (S2|S3,S1), (S2,S3|S1), (S2|S3|S1), (S2,S3,S1), where '|' represents a fusion partition or, in other words, it implies that statements on either side belong to different loop nests. Thus, a total of 24 different fusion partitionings are possible for only 3 statements considered. Similarly, if statements S13-S18 in the second loop nest are considered, there are 90 possible orderings of statements[2], and for each ordering, there are 32 different fusion partitionings, resulting in a total of 2880 possible fusion partitionings. If all the 36 statements of the *swim* benchmark are considered, the search space of possible fusion partitionings becomes unmanageable.

[1]For any two consecutive statements, there exist 2 possibilities - they can either belong to the same loop nest or not. Since, there are a total of (n-1) pairs of consecutive statements for a total of n statements, there exist 2^{n-1} possible fusion partitionings.

[2]The number of orderings of statements is not 6! (=720) because there are dependences among statement pairs S13-S16, S14-S17 and S15-S18, respectively. As a result, some among the 720 total orderings are not legal.

This necessitates the need for an effective cost model to find a good fusion partitioning among all legal partitionings. However, incorporating useful optimization criteria within the polyhedral framework for this purpose is hard as the algorithms involved are computationally intensive (97), and the complexity increases exponentially with the number of statements. Our experiments show that the fusion model employed in the state-of-the-art polyhedral compiler frameworks (98; 99; 87; 100; 101) gives good performance only for kernel programs with few statements, but performs sub-optimally for larger programs such as those in SPEC benchmark suite. We also find that the iterative compilation framework leveraging the polyhedral model (102; 103; 97) cannot construct the search space of legal fusion partitionings for such large programs, as the algorithms employed are not scalable with the number of statements. In this part of the thesis, we propose a new cost model to tackle this important problem that has eluded the polyhedral compiler framework from effectively optimizing large programs.

The cost model employed by our fusion algorithm has 2 objective functions, (1) *maximize data reuse* in the fused loop nest, and (2) *preserve coarse-grained parallelism*, inherent in the source code. In our fusion algorithm, we achieve data reuse through the use of heuristics in a *pre-fusion* step that absolves the polyhedral framework from the responsibility of providing an objective function to guide loop fusion, and keeps the large programs tractable. This pre-fusion step blends effectively with the polyhedral framework and as a result, subsequent benefits of composing multiple loop optimizations can still be achieved. This pre-fusion step also helps us to consider input dependences while evaluating data reuse and not just the true dependences that constitute real edges in the Data Dependence Graph.

The objective of maximizing data reuse, however, conflicts with preserving coarse-grained parallelism that is originally present in the source code. This is because merging multiple statements into the same loop nest may result in a dependence carried by the outer-loop, leading to a loss of outer-loop or coarse-grained parallelism. Thus, in our fusion algorithm, we detect the occurrence of such a dependence between two SCCs and distribute them into separate loop nests such that loss of data reuse is minimized and coarse-grained parallelism is preserved.

Our fusion algorithm is implemented within a source-to-source polyhedral compiler framework, PLuTo. We tested our fusion algorithm, called *wisefuse*, using 10 benchmarks from 3 different benchmark suites, with the large programs from SPEC and NAS Parallel (NP) benchmark suites and the small kernel programs from the Polybench (36) suite. Using our fusion algorithm within the polyhedral framework, we achieve a performance improvement ranging from 1.7X to 7.2X for these large benchmarks over state-of-the-art fusion algorithms leveraging the polyhedral model. We achieve an improvement of 5% to 18% over the Intel compiler for the same benchmarks. Results also demonstrate that our algorithm matches the performance achieved by existing polyhedral frameworks for the kernel programs from Polybench, and in some cases we achieve an improvement of as much as 2.1X due to coarse-grained parallelization.

The rest of the chapter proceeds as follows. Section 6.2 discusses specific background related to loop fusion within the polyhedral framework. Section 6.3 formulates the loop fusion problem and describes the legality constraints for fusion, in the context of the polyhedral framework. Section 6.4 describes our loop fusion algorithm and discusses the heuristics used to achieve the aforementioned objectives. Section 6.5 compares and discusses the performance results of our fusion algorithm and other state-of-the-art algorithms. The related work is presented in Section 6.6 and we conclude in Section 6.7.

6.2 Loop fusion and the polyhedral framework

The polyhedral framework finds legal statement-wise loop hyperplanes one loop-level at a time using an Integer Linear Programming (ILP) solver. If the ILP solver cannot find a solution at the current loop level because of unsatisfied dependences, a *cut* must be issued, represented by a scalar dimension. A *cut* between two SCCs essentially distributes them into different loop nests, thus satisfying certain dependences. The most important criteria in cutting SCCs is based on the dimensionality (depth of the enclosing loop nest) of the SCC, i.e. any two *consecutive* SCCs with different dimensionalities are cut first as they are least likely to aid data reuse. Thus, the initial ordering of the SCCs (we call, the *pre-fusion schedule*[3]) decides which SCCs remain fused and which ones are distributed in the transformed code, and is therefore critical in achieving a good fusion partitioning with effective data reuse.

Loop fusion within the polyhedral framework thus involves three steps

1. finding the Strongly Connected Components (SCCs) in the Data Dependence Graph,

2. determining a legal (and good) pre-fusion schedule, and

3. finding legal statement-wise loop hyperplanes one loop level at a time. This involves

[3]Pre-fusion schedule is called so, because this schedule or ordering of SCCs serves as a guideline for the fusion partitioning obtained in the final transformed program. A *fusion partitioning* is the partitioning of program statements into clusters of statements called *fusion partitions*, where each fusion partition represents a single loop nest.

issuing cuts (represented by scalar dimensions in the affine transform) between SCCs, whose relative ordering is determined in step 2.

SCCs that do not require to be cut to satisfy dependences end up in the same fusion partition after the above 3 steps. Thus, loop fusion is implicitly performed with the finding of the legal loop hyperplanes, in composition with other transformations.

6.2.1 Existing fusion models in the polyhedral framework

The state-of-the-art polyhedral frameworks (PLuTo (98), PoCC (99), PolyOpt (87), LLVM's Polly (104)) employ an effective cost function to find statement-wise loop hyperplanes that minimize communication volume among processors (or, reuse distance in sequential execution). However, there does not exist a useful cost function to find a good pre-fusion schedule that eventually guides loop fusion as explained above. The existing frameworks combine steps 1 and 2, i.e. they use a depth-first traversal of the DDG to find the SCCs and the resultant order also yields a *legal* (but not necessarily *good*) pre-fusion schedule. Existing frameworks do not also employ an optimization criteria for data reuse in step 3 as this would require incorporation of additional constraints in the ILP formulation for finding legal hyperplanes. This may render large programs intractable as these constraints increase exponentially with the number of statements. The pre-fusion schedule obtained from a depth-first traversal of the DDG has 2 drawbacks.

1. It does not attempt to order statements with the same dimensionality consecutively. This leads to sub-optimal fusion. For example, in the *swim* benchmark shown in Figure 6.2, the intermediate statements S4-S12 have a dimensionality of 1, and if any of these statements is ordered (in the pre-fusion schedule) between statements from the first loop nest and statements S13-S18, then a cut to find legal loop hyperplanes would prevent any possible fusion of these statements.

2. It does not order statements with an input dependence between them consecutively, or in other words, it does not consider data reuse through the input or the Read-After-Read (RAR) dependences. This is because the DDG (traversed to find SCCs) does not contain edges corresponding to the input dependences as such edges restrict parallelism. The input dependences are, nonetheless, crucial to achieve effective data reuse. For example, in the *swim* benchmark, if the input dependences between statements S1, S2 and S3 are

not considered, they would not be ordered consecutively in the pre-fusion schedule. As a result, they would be distributed into different loop nests, leading to a loss of data reuse.

Thus, for effective loop fusion, steps 1 and 2 must be treated independently and there must be an effective cost model to determine a good pre-fusion schedule. Lack of a powerful cost model is a serious limitation in the existing polyhedral frameworks as a poorly chosen pre-fusion schedule impacts all the other transformations performed. The performance impact becomes more significant as the number of statements within a SCoP, and consequently, the number of possible pre-fusion schedules becomes larger.

6.3 Problem Formulation

Prior to formulating the problem, we identify the different scenarios that result from fusing two statements, each enclosed in separate loop nests. Here, we consider statements instead of loops, because of the statement-centric view held by the polyhedral compiler. Before fusion, there is either no dependence or a loop-independent dependence between any two statements. Fusion is clearly legal in the former case. In the latter case, however, fusing two statements (S_i and S_j) with a loop-independent dependence can lead to 3 different scenarios.

1. The dependence remains loop independent. In the polyhedral framework, this is represented by the condition,

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) = 0, \langle \vec{s}, \vec{t} \rangle \in P_{e^{S_i \rightarrow S_j}} \tag{6.1}$$

   Clearly, this does not violate the constraint in Equation (2.4) presented in Chapter 2, and thus fusing statements S_i and S_j is legal. Such a dependence allows the loop to be a parallel loop and is called a *precedence* or a *fusion-permitting dependence*.

2. The dependence can become a backward loop-carried dependence. This is represented by the condition,

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) < 0, \langle \vec{s}, \vec{t} \rangle \in P_{e^{S_i \rightarrow S_j}} \tag{6.2}$$

   This dependence violates the constraint in Equation (2.4), i.e. there is at least one instance or loop iteration that does not preserve the direction of the dependence, and thus fusing the two statements is illegal. Such a dependence is called a *fusion-preventing dependence*.

3. the dependence can become a forward loop-carried dependence. This is represented by the condition,

$$\phi_{S_j}(\vec{t}) - \phi_{S_i}(\vec{s}) > 0, \langle \vec{s}, \vec{t} \rangle \in P_{e^{S_i \rightarrow S_j}} \tag{6.3}$$

This dependence satisfies the constraint in Equation (2.4), i.e. it preserves the direction of the dependence, and thus leads to a legal fusion. However, it leads to a forward dependence or non-parallel loop. Although we call it a *precedence dependence* (following previous works), this dependence may also be considered as a *fusion-preventing dependence* if the goal is to preserve parallelism.

With this background on the different types of dependences, we next formulate the loop fusion problem.

In the past, the loop fusion problem has been formulated as a graph partitioning problem (105; 93), where the goal was to partition all loops into disjoint clusters, each cluster representing a set of loops to be fused. However, in the context of the polyhedral compiler framework that takes a statement-centric view, this formulation needs an adjustment in that each vertex of the graph corresponds to a *statement* instead of a *loop*. The relations between statements are represented by a directed graph $G = (V, E = F \cup \bar{F})$, where each statement corresponds to a vertex $v \in V$ of the graph. Edges in E are classified into *fusion-permitting edges* (edges in F) and *fusion-preventing edges* (edges in $\bar{F}$). The problem is thus formulated as one of finding a *fusion partitioning*, i.e partitioning V into clusters of statements that can be legally fused. The partitioning V is legal, subject to the following legality constraints. These constraints together ensure that all dependences are lexicographically positive.

1. Statements in the same SCC belong to the same fusion partition.

2. **Precedence constraint** - The ordering of SCCs in the partitioned graph (that corresponds to the fused program) must respect all dependences between statements.

3. **Fusion-preventing constraint** - The fusion partitioning should not result in any loop-carried backward dependence, i.e. statements connected by an edge in $\bar{F}$ must belong to different fusion partitions.

To ensure the satisfaction of the first constraint, the first step in the fusion algorithm (listed in Section 6.2) involves determining SCCs in the source program. Satisfaction of this constraint

```
for (j = 4; j <= ny+9-2; j++)  // parallel loop
 for (i = 4; i <= nx+9-3; i++)
  for (k = 4; k <= nz+9-3; k++)
    S1: ab[j][i][k] = uyb[j][i][k] * ((a[j-1][i][k] + a[j][i][k])
       + (a[j-2][i][k] + a[j+1][i][k]) + (a[j-3][i][k] + a[j+2][i][k]));

for (j = 4; j <= ny+9-3; j++)  // parallel loop
 for (i = 4; i <= nx+9-2; i++)
  for (k = 4; k <= nz+9-3; k++)
    S2: al[j][i][k] = uxl[j][i][k] * ((a[j][i-1][k] + a[j][i][k])
       + (a[j][i-2][k] + a[j][i+1][k]) + (a[j][i-3][k] + a[j][i+2][k]));

for (j = 4; j <= ny+9-3; j++)  // parallel loop
 for (i = 4; i <= nx+9-3; i++)
  for (k = 4; k <= nz+9-2; k++)
    S3: af[j][i][k] =  uzf[j][i][k] * ((a[j][i][k-1] + a[j][i][k])
       + (a[j][i][k-2] + a[j][i][k+1]) + (a[j][i][k-3] + a[j][i][k+2]));

for (j = 4; j <= ny+9-3; j++)  // parallel loop
 for (i = 4; i <= nx+9-3; i++)
  for (k = 4; k <= nz+9-3; k++)
    S4: athird[j][i][k] = (af[j][i][k+1] - af[j][i][k]) + (al[j][i+1][k] -
         al[j][i][k]) + (ab[j+1][i][k] - ab[j][i][k]) + a[j][i][k];
```

(a)

```
 /*  prologue  */
for (j = lb_j ; j <= ub_j ; j++)
 for (i = lb_i ; i <= ub_i ; i++)
  for (k = lb_k ; k <= ub_k ; k++)  {
    S1: ab[j][i][k] = uyb[j][i][k] * ( . . );
    S2: al[j][i][k] = uxl[j][i][k] * ( . . );
    S3: af[j][i][k] = uzf[j][i][k] * ( . . );
                                    backward
                                    dependence
    S4: athird[j][i][k] = (af[j][i][k+1] - af[j][i][k]) + . . ;
  }
 /*  epilogue  */
```

(b)

```
 /*  prologue  */
for (j = lb_j ; j <= ub_j ; j++)  // fwd-dep loop
 for (i = lb_i ; i <= ub_i ; i++)  // fwd-dep loop
  for (k = lb_k ; k <= ub_k ; k++)  {
    S1: ab[j][i][k] = uyb[j][i][k] * ( . . );
    S2: al[j][i][k] = uxl[j][i][k] * ( . . );
S3:     af[j][i][k] = uzf[j][i][k] * ( . . );
S4:     athird[j-1][i-1][k-1] = (af[j-1][i-1][k]
         - af[j-1][i-1][k-1]) + . . . ;
forward
dependence
  }
 /*  epilogue  */
```

(c)

Figure 6.3: (a) Original source code for *advect*; (b) Fully fused *advect* without shifting (incorrect code); (c) Fully fused *advect* with shifting (correct code)

is ensured by ordering SCCs instead of statements in the further steps that achieve some optimization criteria. The ordering of SCCs must satisfy the precedence constraint to ensure that a legal partitioning exists that leads to correct execution. Thus, the pre-fusion schedule that decides the initial ordering of SCCs (step 2 in the fusion algorithm) must satisfy the precedence constraint. However, a pre-fusion schedule that satisfies the precedence constraint may still lead to a partitioning where there is a loop carried backward dependence between two statement instances in a partition. Such a partitioning is not legal. For example, fusing all four statements of the *advect* benchmark as shown in Figure 6.3(b), leads to a loop-carried backward dependence, although the precedence constraint is satisfied.

In such a scenario, the polyhedral framework would either apply the *loop shifting* transformation to remove the backward dependence or issue a cut between statements to satisfy the dependence. In other words, *if the pre-fusion schedule satisfies the precedence constraint, then the polyhedral framework guarantees legal fusion partitioning that satisfies the fusion-preventing constraint.*

In case of the *advect* benchmark, legality is ensured through shifting S4 by 1 iteration and thus removing the backward dependence as shown in Figure 6.3(c). As a result of applying the shifting transformation, the backward dependence from S4 to statements S1-S3 turns into a forward dependence from statements S1-S3 to S4. This renders the outer loops i and j as forward dependence loops, or in other words, coarse-grain parallelism is lost as shown in Figure 6.3(c).

Thus, the problem of loop fusion within the polyhedral framework can be formulated as one of finding a pre-fusion schedule that satisfies the precedence constraint. In addition to legality, the criteria for deciding good partitions is two-fold, maximizing *data reuse* and *parallelism*. In the following section, we propose a fusion algorithm that achieves precisely these two key objectives in the polyhedral framework.

6.4 Our Fusion Algorithm

As explained in previous sections, a pre-fusion schedule or the initial ordering of the SCCs has a significant bearing on the fusion partitioning achieved in the transformed program. In this section, we first describe our algorithm (Algorithm 3) for finding a good pre-fusion schedule and the heuristics underlying the algorithm. We use the same example of the *swim* benchmark shown in Figure 6.2 to elucidate our algorithm. We next describe Algorithm 4, which is used to achieve the second objective of preserving coarse-grained parallelism in the transformed code. We use the example of the *advect* benchmark introduced in Section 6.3 for the purpose of explanation.

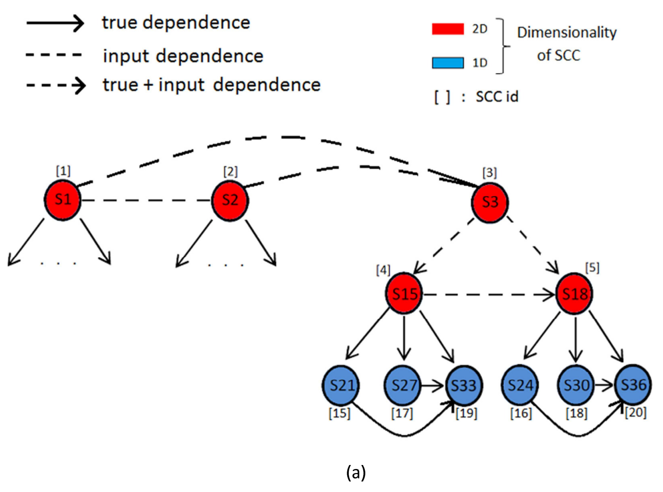
6.4.1 Algorithm 3: Finding a good pre-fusion schedule

A good pre-fusion schedule is one that orders the SCCs so that, SCCs with significant reuse between them are merged in the same loop nest in the transformed code. In order to achieve a good pre-fusion schedule, the ordering of SCCs is done based on the following criteria:

- **Constraint:** The ordering must respect the precedence constraint among the SCCs
- **Heuristic 1:** SCCs that allow data reuse and have the same dimensionality are ordered consecutively
- **Heuristic 2:** SCCs are considered for re-ordering in the original program order

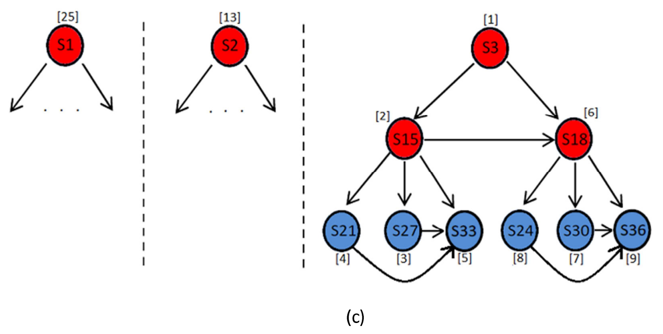
The above criteria does not necessarily lead to a fusion partitioning that maximizes data reuse, but it does lead to one that has significant data reuse.

Rationale for the chosen heuristics. As explained in Section 6.3, even with the satisfaction of the precedence constraint, there may be fusion preventing dependences between statements and hence some SCCs will need to be cut to satisfy those dependences. The heuristics chosen



```
for (i=0; i<M; i++) {
 for (j=0; j<N; j++) {
  S1: UNEW[i+1][j] = .. CV[i][j] .. H[i][j] ..
  S2: VNEW[i][j+1] = .. CU[i][j] .. H[i][j] ..
  S3: PNEW[i][j] = POLD[i][j] .. CU[i][j] .. CV[i][j] ..
  S15: POLD[i][j] = P[i][j] .. PNEW[i][j] ..
  S18: P[i][j] = PNEW[i][j];   } }
. .
for (i=0; i<M; i++) {
 for (j=0; j<N; j++) {
 S13: UOLD[i][j] = U[i][j] ..
 S16: U[i][j] =  UNEW[i][j];
 S14: VOLD[i][j] = V[i][j] ..
 S17: V[i][j] =  VNEW[i][j];   } }
. .
```

(b)



```
for (i=0; i<M; i++) {
 for (j=0; j<N; j++) {
  S3: PNEW[i][j] = POLD[i][j] .. CU[i][j] .. CV[i][j] ..
  S15: POLD[i][j] = P[i][j] .. PNEW[i][j] ..   } }
. .
for (i), for (j)
  S18: P[i][j] = PNEW[i][j];
. .
for (i), for (j)
  S2: VNEW[i][j+1] = .. CU[i][j] .. H[i][j] ..
. .
for (i), for (j)
   S14: VOLD[i][j] =  V[i][j] ..
. .
for (i), for (j)
   S17: V[i][j] =  VNEW[i][j];
. .
for (i), for (j)
  S1: UNEW[i+1][j] = .. CV[i][j] .. H[i][j] ..
. .
for (i), for (j)
   S13: UOLD[i][j] =  U[i][j] ..
. .
for (i), for (j)
   S16: U[i][j] =  UNEW[i][j];
. .
```

(d)

Figure 6.4: Our fusion model: (a) Partial DDG for *swim* and pre-fusion schedule from Algorithm 3; (b) Fused code corresponding to (a); (c) Pre-fusion schedule chosen by PLuTo; (d) Fused code corresponding to (c)

are such that they aim to minimize loss of reuse through these inevitable cuts. Since the first and the most important criteria of issuing a cut is based on the dimensionality of the SCCs, *heuristic 1* ensures that SCCs that have reuse remain immune to a cut with high probability. Also, since data reuse through input dependences is considered, we achieve better overall data reuse. This is possible mainly because we decouple the pre-fusion scheduling from the step where the DDG (with only true dependences) is traversed to find the SCCs. *Heuristic 2* allows more SCCs to be fused together into a single loop nest. The reason is that, if SCCs close to each other in original program order are fused, then it increases the probability of following SCCs to become legally fusable because the precedence constraint is satisfied. Also, SCCs that are close to each other tend to have higher reuse through reads to the same data items.

ALGORITHM 3: Finding a good pre-fusion schedule

```
1: INPUT:
   List of statements in the program: $S(1, ..., n)$
   Adjacency matrix representing true dependences: $adj$(1,..,n)(1,..,n)
   Adjacency matrix representing input dependences: $RARadj$(1,..,n)(1,..,n)
   List of statements belonging to an SCC that contains statement $s$: $SCC_s$
2: begin
3: Initialize $id$ to 0
4: for each statement $s \in S$ do
5:    $visited(s)$ = 0
6: for each statement $s \in S$ do
7:    if $visited(s)$ = 0 then
8:       $fusable = \emptyset$ {/* where $fusable$ is the list of statements, fusable with statement $s$ */}
9:       for each statement $t \in SCC_s$ do
10:         $visited(t) = 1$
11:         $sccId(t) = id$
12:         Add $t$ to $fusable$
13:      Increment $id$ by 1
         {/* Lines 16, 17, 18 check for dimensionality, reuse and precedence constraint resp. */}
14:      for each statement $t \in S$ s.t. $visited(t)$ = 0 $\wedge$ $dimension(s)$ = $dimension(t)$ do
15:         if $\exists\ i \in fusable$, s.t. $adj(i, j) = 1 \vee RARadj(i, j) = 1, \forall\ j \in SCC_t$ then
16:            if $\nexists\ s' \in S$, s.t. $adj(s',t') \neq 0, \forall\ t' \in SCC_t$ then
17:               for each statement $t' \in SCC_t$ do
18:                  $visited(t') = 1$
19:                  $sccId(t') = id$
20:                  Add $t'$ to $fusable$
21:               Increment $id$ by 1
22: end
23: OUTPUT: Pre-fusion Schedule
```

Explanation of the algorithm. In order to find a pre-fusion schedule, it is essential that the Strongly Connected Components are first extracted from the Data Dependence Graph. This is done using the Kosaraju's algorithm (106) as in existing polyhedral compiler frameworks. But, our fusion algorithm determines the pre-fusion schedule in a separate step. The choice of the pre-fusion schedule is made to meet the above-mentioned three criteria. For this purpose, Algorithm 3 begins a traversal of the DDG in program order. For a statement s thus 'visited', Algorithm 1 finds an 'unvisited' statement t (i.e. $visited(t) = 0$) with the same dimension and that also has data reuse with statements already marked as *fusable* with statement s. The algorithm further checks to see whether t satisfies the precedence constraint. The precedence constraint is satisfied if no statement t' belonging to the SCC that contains t (SCC_t) depends on an 'unvisited' statement node s', i.e. $adj(s',t') \neq 0$. If the precedence constraint is thus satisfied, all statements belonging to SCC_t are marked visited, assigned the next higher id than SCC_s and put into the set, *fusable*. This ensures that the reordering of SCCs achieved satisfies all 3 criteria. The heuristics used can be seen in play for the *swim* benchmark in Figure 6.4.

Figure 6.4 compares the pre-fusion schedule obtained from Algorithm 3 with that achieved by the state-of-the-art pre-fusion scheduling algorithm used in PLuTo. The other existing polyhedral compilers such as PoCC (99), PolyOpt (87), LLVM's Polly (104) all use the fusion algorithm described by Bondhugula et al. in their paper (107) and implemented in their tool, PLuTo. The fusion algorithm employed in PLuTO uses a depth-first traversal to find and order the SCCs. Figures 6.4(a) and (c) show the partial DDG for the same code excerpt of the *swim* benchmark shown earlier in Figure 6.2. The values in square brackets are the SCC *id* for the 36 statements of the *swim* benchmark, and are representative of the pre-fusion schedules chosen by Algorithm 3 and PLuTo, respectively. Since Algorithm 3 also considers reuse through the input dependences, the DDG in (a) also shows input dependences marked with dashed lines. Figures 6.4(b) and (d) show the transformed codes for *swim* generated from the pre-fusion schedules given by Algorithm 3 and PLuTo, respectively. From the figures, following observations can be made about the pre-fusion schedule obtained from Algorithm 3.

1. Statement S18 is scheduled immediately after statement S15 as there is an opportunity for reuse and the precedence constraint is satisfied. Since both S15 and S18 have the same dimensionality, they remain fused in the transformed code shown in Figure 6.4(b). This opportunity for reuse is missed by the fusion model used in PLuTo since statement S27 (SCC$_{id}$=3) with a different dimensionality is scheduled immediately after S15 (SCC$_{id}$=2),

as shown in Figure 6.4(c).

2. Reuse through input dependences is considered. As a result, statements S1, S2 and S3 that have reuse through input dependences and the same dimensionality are ordered next to each other as shown in (a). This allows them to be fused in the transformed code as shown in (b). Again, this opportunity for significant data reuse is missed by PLuTo. It schedules these 3 statements separately because they are disconnected in the DDG as shown in (c).

3. Ordering SCCs that are close to each other in program order consecutively, allows more statements to be fused together in the same loop nest because of satisfaction of precedence constraint. For example, statements S1, S2 and S3 were considered in program order. Since S3 was already scheduled, it created the opportunity for statements S15 and S18 to also be fused in the same loop nest because the precedence constraint was satisfied. As a result, 5 statements could be fused together in the same loop nest as shown in (b), thus allowing significant data reuse. In contrast, a maximum of 2 statements are fused in a loop nest using the fusion model in PLuTo as shown in (d).

4. Statements S13, S16 and statements S14, S17 could not be fused with the other statements in the first loop nest, as these statements depend on the intermediate statements (S4-S12, that update the variables at grid boundary). In other words, scheduling these statements with the other statements violates the precedence constraint and thus are distributed into a different loop nest.

6.4.2 Algorithm 4: Enabling Outer-level Parallelism

As explained in Section 6.2, in the polyhedral framework, legal statement-wise hyperplanes are found one loop-level at a time using an ILP solver. The cost function used aims to find a loop hyperplane that minimizes the communication traffic among processors. As a result, it first aims to find a hyperplane that involves no communication, i.e. an outer-parallel loop. If no communication-free hyperplane can be found, a pipelined-parallel (or, forward dependence) hyperplane with constant communication cost is found. This was seen in the case of *advect* benchmark as shown in Figure 6.3(c).

The pre-fusion schedule obtained from Algorithm 3 leads to a partitioning that is legal and achieves good reuse. However, this may deprive the transformed code of outer-loop parallelism because of the introduction of a forward dependence at the outermost loop, between two of the fused statements. Although, with the outermost loop becoming a forward-dependence loop, pipelined parallelism can still be achieved (for example, in the *advect* benchmark), but the performance can be far from optimal because of increased communication costs. Algorithm 4 thus aims to preserve parallelism in the outermost loop to enable a coarse-grained parallel code, although at a cost of some loss of data reuse.

ALGORITHM 4: Enabling outer-level parallelism

```
INPUT:
List of dependences in the program: $deps$
begin
$\{$For the first non-serial loop hyperplane, $h$, found by the ILP solver$\}$
for each dependence, $S_i \to S_j, \in deps$ do
  if $S_i \to S_j$ is not satisfied at loop-level $h \wedge \phi^h_{S_j}(\vec{t}) - \phi^h_{S_i}(\vec{s}) > 0, \langle \vec{s}, \vec{t} \rangle \in P_{e_{S_i \to S_j}}$ then
    Issue a cut between SCCs containing statements $S_i$ and $S_j$
    Discard the found hyperplane, and solve for a new hyperplane with the updated DDG
end
OUTPUT: Fusion with possible outer-level parallelism
```

For the pre-fusion schedule determined through Algorithm 3, the ILP solver starts finding legal loop hyperplanes. For the first non-serial[4] loop hyperplane thus found, Algorithm 4 checks for any unsatisfied forward-dependence at that loop level. A forward dependence between two statement instances is determined through the inequality in Equation 6.3. An existence of a forward-dependence implies a forward-dependence outermost loop. To satisfy the dependence and thus to generate a parallel outermost loop, we issue a cut between the dependent SCCs, and re-solve for a new hyperplane with the updated dependence information in the DDG. This may be repeated until all dependences are satisfied and outer-loop parallelism is restored. *Since we issue the cut between SCCs carrying the actual dependence and not arbitrarily, the transformed code is minimally distributed, or in other words, it suffers minimal loss of data reuse.* This can be seen in the transformed code for the *advect* benchmark generated from Algorithm 4 as shown in Figure 6.5. Statements S1, S2 and S3 are still merged in the same loop nest and enjoy reuse through reads, while only statement S4 is distributed into a different loop nest. Thus, coarse-grained parallelism is achieved with a minimal loss of reuse. The existing polyhedral

[4]a serial loop hyperplane cannot be parallelized

```
for (t2=4; t2<=ny+6; t2++)  // parallel loop
 for (t3=4;t3<=nx+6;t3++)
  for (t4=4;t4<=nz+6;t4++) {
    S1: ab[t2][t3][t4]= uyb[t2][t3][t4] * ( .. a[j][i][k] .. );
    S2: al[t2][t3][t4]= uxl[t2][t3][t4] * ( .. a[j][i][k] .. );
    S3: af[t2][t3][t4]= uzf[t2][t3][t4] * ( .. a[j][i][k] .. );
  }

/*  Update of variables at grid boundary  */

for (t2=5; t2<=ny+7; t2++)  // parallel loop
 for (t3=5;t3<=nx+7;t3++)
  for (t4=5;t4<=nz+7;t4++)
    S4: athird[t2-1][t3-1][t4-1] = (af[t2-1][t3-1][t4] - af[t2-1][t3-1][t4-1]) + . . . ;
```

Figure 6.5: Transformed *advect* code generated using Algorithm 4

frameworks, on the other hand, perform maximal fusion after loop shifting leading to loss of coarse-grained parallelism as shown in earlier in Figure 6.3(c).

6.5 Experimental Evaluation

We implemented our fusion algorithm within PLuTo. We replaced its algorithm for finding the pre-fusion schedule with ours, and also incorporated our algorithm for preserving the coarse-grained parallelism. However, for the purpose of finding loop hyperplanes, we rely on the same cost function as used in PLuTo and described in (108).

6.5.1 Setup

Fusion Model	Description
Intel Compiler	Fusion within Intel compiler (baseline)
wisefuse	Our fusion model
smartfuse	The default fusion model in PLuTo. It uses heuristics to 'cut' or distribute SCCs into different loop nests
nofuse	Another fusion model implemented in PLuTo. It separates all SCCs into different loop nests.
maxfuse	Another fusion model implemented in PLuTo. It conservatively distributes SCCs into different loop nests

Table 6.1: Summary of the fusion models

We ran our experiments on an Intel Xeon processor (E5-2650) with 8 Sandy Bridge-EP cores, operating at 2.0GHz. The processor has private L1 (32KB per core) and L2 (256KB per core) caches and a 20MB shared L3 cache. The experimental results compare the performance between our fusion algorithm and the one used within PLuTo that employs three different fusion heuristics. We compare against all the three heuristics used. The reason for comparing with PLuTo, as explained before, is that various other automatic polyhedral frameworks such as (99; 87; 104) use the same fusion algorithm as in PLuTo, and others (101; 109) use its variants. Since PLuTo cannot parse Fortran code, we integrated our fusion model with PolyOpt/Fortran (87), a tool that uses ROSE compiler (88) frontend to parse Fortran code and relies on PLuTo to accomplish loop fusion. A summary of the different fusion models compared in this section is given in Table 6.1. The transformed code generated using different fusion models was compiled using the Intel compiler v13 (icc and ifort) as the backend compiler. The compile time options used with the Intel compiler were '-O3' and '-parallel'. The original source, automatically parallelized using the Intel compiler and with all high level optimizations including loop fusion enabled, was used as the base case for comparison.

6.5.2 Benchmarks

The benchmarks used in the experiments belong to three different benchmark suites. We chose large programs from the SPEC and the NAS Parallel (NP) benchmark suites. The small kernel programs were chosen mainly from Polybench and PLuTo. These are summarized in Table 6.2. In the table, the first five benchmarks correspond to the large programs, and the last five represent small kernel programs.

Benchmark	Benchmark Suite	Category	Problem Size
gemsfdtd	SPEC 2006	Computational Electromagnetics	Reference Input
swim	SPEC OMP	Shallow Water Modeling	Reference Input
applu	SPEC OMP	Computational Fluid Dynamics	Reference Input
bt	NPB	Block Tri-diagonal solver	CLASS C; $(162)^3$, dt = 0.0001
sp	NPB	Scalar Penta-diagonal solver	CLASS C; $(162)^3$, dt = 0.00067
advect	PLuTo	Weather modeling	nx=ny=nz=300
lu	Polybench	Linear Algebra	N=1500
tce	Polybench	Computational Chemistry	Standard; $(55)^3$
gemver	Polybench	Linear Algebra	N=1500
wupwise	SPEC OMP	Quantum Chromodynamics	Reference Input

Table 6.2: Summary of the benchmarks

6.5.3 Results and Discussion

Figure 6.6 shows the normalized performance with respect to the Intel compiler achieved using different fusion models on the different benchmarks considered. Each of the codes was run using all 8 cores on the test processor. The figure also shows the geometric mean (represented as GM in Figure 6.6) of the performance achieved by different fusion models over the 10 benchmarks - *wisefuse* achieves a performance improvement of 1.3X over the Intel compiler.

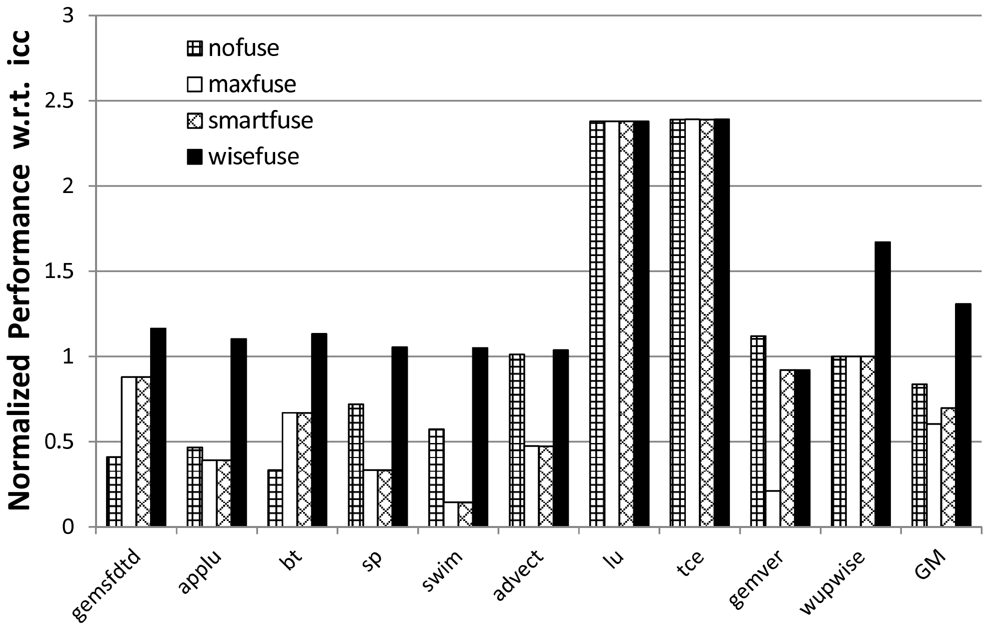
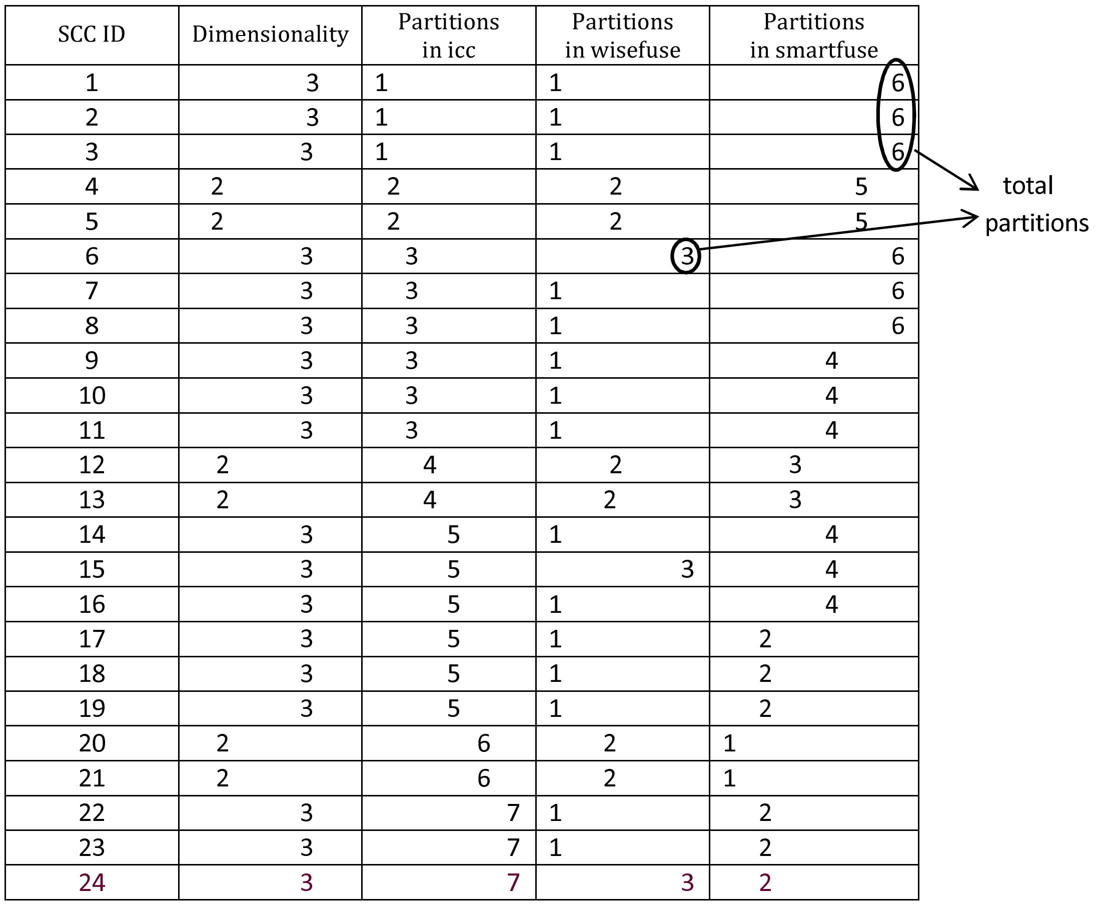


Figure 6.6: Results

For the purpose of discussing our experimental results, we divide the benchmarks into three categories, (1) the large benchmark programs, (2) the programs where fusion leads to a loss of parallelism, and (3) small kernel programs.

Large Benchmark Programs. Large programs tend to have multiple loop nests, with the potential to serve as ideal playground for the polyhedral compilers. However, ironically, none of the polyhedral frameworks has yet demonstrated success with large programs. As explained earlier, a fundamental reason for this is the lack of an effective cost model for loop fusion, which in turn hurts all other transformations.

Figure 6.6 shows that our fusion algorithm, *wisefuse*, achieves a performance improvement in the range of 1.7X to 7.2X as compared to *smartfuse* for the large programs, and an improvement of 5-18% over the Intel Compiler. It is important to note that This performance improvement is attributed to the better fusion partitioning achieved by *wisefuse*, with the amount of improvement achieved being a function of the reuse opportunity available in the source program. For large programs, only a part of the program usually provides fusion opportunity. For example, *wisefuse* improves UPMLupdatee and UPMLupdateh routines in *gemsfdtd* benchmark by nearly 1.5X, but the overall program by only 1.18X. Similar is the case with APPLU, SP and BT benchmarks. The following discussion provides further insight into the results obtained.


total
partitions

SCC ID	Dimensionality	Partitions in icc	Partitions in wisefuse	Partitions in smartfuse
1	3	1	1	6
2	3	1	1	6
3	3	1	1	6
4	2	2	2	5
5	2	2	2	5
6	3	3	3	6
7	3	3	1	6
8	3	3	1	6
9	3	3	1	4
10	3	3	1	4
11	3	3	1	4
12	2	4	2	3
13	2	4	2	3
14	3	5	1	4
15	3	5	3	4
16	3	5	1	4
17	3	5	1	2
18	3	5	1	2
19	3	5	1	2
20	2	6	2	1
21	2	6	2	1
22	3	7	1	2
23	3	7	1	2
24	3	7	3	2

Figure 6.7: Partitioning achieved by different fusion models for the *gemsfdtd* benchmark (values in each column are spaced out for readability)

Figure 6.7 compares the fusion partitioning achieved by icc, *smartfuse* and *wisefuse*, for the *UPMLupdateh* subroutine in the *gemsfdtd* benchmark. The *UPMLupdateh* and another similar subroutine, *UPMLupdatee*, together account for 45% of the total execution time in *gemsfdtd*. In the figure, the second column shows the dimensionality of each SCC within *UPMLupdateh*, and columns 3 through 5 show the partition number to which each SCC belongs to, in the transformed codes obtained from different fusion strategies. The SCCs with the same partition number in the figure are fused in the same loop nest in the transformed code. It can be seen that *wisefuse* minimizes the number of partitions and thus, in effect achieves maximum reuse. All the SCCs fused together have reuse not only through true dependences but also through input dependences. The key to fusing more SCCs together is that *wisefuse* chooses a pre-fusion schedule such that SCCs with the same dimensionality are ordered next to each other, increasing the probability for them to be fused together into perfect loop nests. For example, all SCCs with a dimensionality of 2 are perfectly fused in the transformed *gemsfdtd* code generated using *wisefuse*.

Also, *wisefuse* employs another heuristic - it fuses SCCs close to each other in program order as such SCCs tend to reuse more data and this also leads to effective fusion. This was observed in *applu*, *bt* and *sp* benchmark applications, where *wisefuse* fused SCCs that belonged to the same pass (x-, y- or z-pass) and thus enjoyed excellent reuse through the input dependences. *Smartfuse* on the other hand, fused statements across different passes, which although led to reuse through the true dependences, deprived it from achieving better reuse among the SCCs close in program order. This led to overall poor reuse achieved through *smartfuse* in such programs.

For all the large programs, *icc* largely maintains the original program order and doesn't accomplish any fusion. This is because *icc* performs a pair-wise fusion (96), where pairs of loops are fused incrementally to prevent exponential cost of finding all fusion choices. However, when successive loops are of different dimensionality as in the *gemsfdtd* benchmark, *icc* does not fuse them. This is because fusing loop nests of different dimensionality does not lead to significant reuse and the fused nest may contain conditional statements that hinders compiler auto-vectorization. *Wisefuse*, instead, reorders statements such that those with (reuse and) the same dimensionality are ordered consecutively in a pre-fusion step for them to be considered for fusion. This allows us to achieve both global reuse (as seen in Figure 6.6) and is effectively vectorized by the backend compiler. As compared to *icc*, *smartfuse* generally performs worse due to lack of a cost model to determine a good pre-fusion schedule.

Programs with Conflict between Fusion and Parallelism. In the case of *advect* and *swim* benchmarks, a maximal fusion leads to a loss of outer-loop parallelism. In these benchmarks, *wisefuse* detects the SCCs that carry a forward dependence on the outer-loop and selectively distributes them to different loop nests as was shown earlier in Figure 6.5 for the *advect* benchmark. Both *smartfuse* and *maxfuse* fusion strategies apply maximal fusion in these cases resulting in a loss of outer-loop parallelism. Although the transformed codes generated by *smartfuse* and *maxfuse* fusion strategies are pipelined parallel, they perform worse than *wisefuse* because of the constant communication costs involved after the parallel execution of each wavefront. It is for the same reason that the transformed code generated by *wisefuse* scales better than *smartfuse*, and the performance gap increases with the increase in the number of processors. *Wisefuse* also slightly outperforms *icc* (that achieves no fusion) because of the better reuse accomplished in each case.

Small Kernel Programs. Among the various small kernel programs from the Polybench benchmark suite, we chose *lu* and *tce* benchmarks as these two benchmarks particularly provide an opportunity for fusion and also reveal the limitations of the current non-polyhedral compilers. The *lu* benchmark implements the Gaussian elimination algorithm that converts a system of linear equations to a unit upper-triangular system. Thus, the *lu* benchmark exhibits a non-rectangular iteration space, or in other words, non-conformable loop bounds. As a result, *icc* adopts a conservative approach and does not achieve coarse-grained parallelization. The polyhedral frameworks, on the other hand, armed with exact dependence information, are able to find available parallelism even in benchmarks with non-rectangular iteration spaces such as *lu*. Both *smartfuse* and *wisefuse* achieve the same fusion partitioning and consequently the same performance, that is considerably better than *icc*.

The *tce* kernel appears in computational quantum chemistry problems. It contains 4 loop nests with significant reuse opportunity among statements. However, since each loop nest contains loops in different order, the non-polyhedral compilers such as *icc* cannot find a conformable pattern to accomplish loop fusion. The polyhedral compilers, on the other hand, find common loop hyperplanes for the different statements and thus achieve loop fusion. Again, both *wisefuse* and *smartfuse* yield similar fusion partitions. Similarly, for other benchmarks from the Polybench benchmark suite, *wisefuse* achieves the same fusion partitioning as *smartfuse*, proving the effectiveness of the heuristics employed by *wisefuse* even for small kernel programs.

The *gemver* benchmark considered earlier shows interesting behavior with different fusion models. Both *wisefuse* and *smartfuse* achieve identical fusion partitioning. However, they perform worse than both the *icc* and *nofuse* fusion models for the chosen problem size. The Intel compiler, like the *nofuse* fusion model does not accomplish any fusion, i.e. all 4 statements in *gemver* are left unfused. *Wisefuse* and *smartfuse* fuse statements S1 and S2, but at the cost of spatial locality in the two statements. *Nofuse* outperforms *icc* as *icc* fails to achieve coarse-grained parallelism in the loop nest enclosing statement S2. It is for this reason that the fusion partitioning accomplished by *wisefuse* and *smartfuse* is still better than that achieved by *icc* as loss of coarse-grained parallelism in even one loop nest hurts scalability when the number of processors are increased.

Among small programs, we also chose the *wupwise* benchmark. Although from the SPEC OMP benchmark suite, 60% of its execution time is spent executing the *zgemm* subroutine, which is nothing but complex matrix-matrix multiplication. However, within the SPEC suite, it is written as a collection of imperfect nests (loop nests of different dimensionality) and also involves data-dependent control flow. Although a recent work (110) has made data-dependent control flow amenable for the polyhedral frameworks through the use of predicates, no available implementation exists for the same. We implemented the predication strategy as proposed in (110) and were thus able to optimize the *wupwise* benchmark. Since *wupwise* consists of imperfect nests, *wisefuse* distributes them into different perfect loop nests so as to achieve better data reuse. This results in a serial performance improvement of 20% over *icc*, that avoids loop distribution given imperfect nests. However, interestingly, the performance improvement goes up to 40% for 8 cores. This was because, loop distribution allowed it to selectively parallelize loop nests, as not all loops nests contain enough computation to benefit from the parallelization. Thus, this proved to be an additional advantage of using the heuristic that favors the fusion of SCCs with the same dimensionality.

6.6 Related Work

In this section, we present the related work on loop fusion in 2 phases - the work done prior to the advent of the polyhedral compilers, and work done within the domain of polyhedral compiler framework.

Loop fusion has been extensively studied in the compiler community as an effective optimization to improve data reuse since its advent in 1980's. Since then, loop fusion has been studied to achieve complementary goals of preserving parallelism, minimizing register pressure and minimizing synchronization. Kennedy et al. in (93) study loop fusion to preserve parallelism while minimizing synchronization. They also show that finding fusion partitions that maximize data reuse is an NP-hard problem, and provide a polynomial time solution to improve data reuse. In their formulation, Kennedy et al. represent data reuse by an edge between loops (nodes), whereas Ding and Kennedy (96) later gave another formulation based on the hyper graph where the computation units are represented as nodes, and each data array is represented as a hyper edge connecting all nodes where the array is accessed. These works thus treated the two problems of preserving parallelism and maximizing reuse independently. Singhai and McKinley in (94) consider these two problems together and use heuristics to find solutions that achieve good reuse and preserve parallelism. Megiddo and Sarkar in (95) target the same problem and show that optimal solutions can be found using their proposed integer programming formulation for problem sizes that occur in practice. In both of these works, the parallel loop nests are already identified and the problem reduces to merging loop nests such that parallelism is not hurt. This decoupling of loop fusion and parallelism misses certain good solutions. In addition, fusion and parallelism are sometimes enabled only after other loop transformations such as skewing, interchange, shifting, etc. are applied. This composition of high-level transformations is not facilitated within the traditional compiler frameworks, leading to the development of the polyhedral compiler framework.

The state-of-the-art fully automatic fusion algorithm (107) within the domain of polyhedral compilers is employed in the source-to-source compilation tool, PLuTo (98). It subsumes previous work in affine scheduling (111; 112; 113; 114) and partitioning (115; 116) and overcomes their major limitation by incorporating an effective cost model for coarse-grained parallelization and data locality. The fusion algorithm employed within PLuTo is also used by various other polyhedral frameworks including PoCC, PolyOpt, LLVM's Polly and IBM's XL compiler. However, as discussed in our motivation to this work, it lacks a cost model to determine good pre-fusion schedules leading to suboptimal fusion partitions, especially for large programs with many statements. Recently, the fusion algorithm used in PLuTO was revised (117), that also aims to preserve coarse-grained parallelization while allowing loop fusion. However, their approach, in addition to relying on a suboptimal pre-fusion schedule, may lead to significant loss of reuse in order to preserve parallelism. This is because, unlike *wisefuse* that precisely distributes only the SCCs preventing parallelism to minimize loss of reuse, the proposed approach in (117) uses heuristics and may end up in excessive distribution to restore parallelism.

In order to find the best fusion partitioning, the authors in (102; 103; 97) have proposed iterative compilation framework leveraging the polyhedral model. The authors propose techniques to build a convex space of all legal and non-redundant fusion partitions, and an optimization algorithm to explore this space. This technique has been shown to find the optimal fusion schedule for small benchmark programs through iterative search. However, we observed that the iterative compilation framework fails to build the search space for even moderately sized programs because the search space grows exponentially with the size of the program. We propose a fusion algorithm with a static cost model to achieve data reuse and preserve parallelism and thus helps to overcome this limitation, especially for programs with many statements.

In addition, URUK/WRaP-IT (118; 119) and CHiLL (39) are other well known tools that use the polyhedral representation to perform various high level optimizations. However, both of these tools are semi-automatic, and require transformations to be specified manually by an expert. The authors of URUK in (118) also achieve performance improvement for some large programs from the SPEC benchmark suite. Our fusion algorithm, on the other hand, demonstrates performance gains on such large programs through a fully automatic application of polyhedral techniques by incorporating an effective cost model for loop fusion.

6.7 Conclusion

In this work, we presented a loop fusion algorithm, *wisefuse*, with 2 objective functions - maximizing data reuse and preserving parallelism. To achieve data reuse, *wisefuse* employs certain heuristics in a pre-fusion step that work in consonance with the polyhedral framework and help to find good fusion partitions. In addition, *wisefuse* ensures that coarse-grained parallelism inherent in the source code remains preserved as aggressive loop fusion may lead to a loop-carried dependence on the outer-loop that prevents parallelism. Experimental results demonstrate that *wisefuse* achieves good fusion partitions not only for the small kernel programs but also the large benchmark programs. *Wisefuse* thus allows to overcome a major limitation in the existing polyhedral frameworks in handling large programs.

Chapter 7

Addressing Scalability: Optimizing Large Programs at an Easy Price

7.1 Introduction

Compiler scalability is a well known problem: precise analysis of large program scopes to reason the correctness of optimizations leads to an exponential increase in the compile time and memory requirement. As a result, production compilers choose to limit optimization to small scopes. In other words, they trade performance for compile time (and programmer productivity (120)). However, with the onset of the era of multiple (many) cores of chip and corresponding accentuation of the memory and bandwidth wall, there is renewed focus on compiler optimizations for improving the memory performance. These optimizations particularly include temporal locality enhancing optimizations such as loop fusion, and other supporting optimizations such as loop interchange and shifting. As a result, analyzing large program scopes to exploit data reuse opportunities spanning multiple nests of loops, or in other words, performing global program transformations, is necessitated. In fact, our experiments and previous work confirm the immense potential for improving the memory (and hence, parallel) performance of applications through such global program transformations.

Previous work on compiler scalability has focused upon either reducing the cost of analyzing each dependence or the number of dependences to be analyzed, or both. The compilers perform such an analysis to find the Program Dependence Graph (PDG) (121). The PDG can then be used further to reason the application of program optimizations. However, it is this step

```
for(i=0;i<N;i++) {
 for(j=0;j<N;j++) {

  for(k=0;k<N;k++) {
   S1: c1 = 0.5 * rho[i][j][k];
   S2: flux1[k] = c2 * u[i][j][k];
   S3: flux2[k] = c1 * u[i][j][k]; }

  for(k=1;k<N-1;k++) {
   S4: rsd1[i][j][k] = rsd1[i][j][k] - c3 * (flux1[k+1] - flux1[k-1]);
   S5: rsd2[i][j][k] = rsd2[i][j][k] - c4 * (flux2[k+1] - flux2[k-1]); }

}}

for(i=0;i<N;i++) {
 for(k=0;k<N;k++) {

  for(j=0;j<N;j++) {
   S6: c5 = 0.5 * rho[i][j][k];
   S7: flux1[j] = c6 * u[i][j][k];
   S8: flux2[j] = c5 * u[i][j][k]; }

  for(j=1;j<N-1;j++) {
   S9: rsd1[i][j][k] = rsd1[i][j][k] - c7 * (flux1[j+1] - flux1[j-1]);
   S10: rsd2[i][j][k] = rsd2[i][j][k] - c8 * (flux2[j+1] - flux2[j-1]); }

} }
```

(a)

```
for(i=0;i<N;i++) {
 for(j=0;j<N;j++) {

  for(k=0;k<N;k++) {
   c1, flux1[k] = c1  }

  for(k=1;k<N-1;k++) {
   rsd1[i][j][k] = rsd1[i][j][k], flux1[k+1], flux1[k-1] }

}}

for(i=0;i<N;i++) {
 for(k=0;k<N;k++) {

  for(j=0;j<N;j++) {
    c5, flux[j] = c5 }

  for(j=1;j<N-1;j++) {
   rsd[i][j][k] = rsd1[i][j][k], flux1[j+1], flux1[j-1] }

}}
```

(b)

Figure 7.1: (a) Original program; (b) Representative program with condensed set of statements and dependences

of finding optimization, especially when it involves large scopes such as in loop fusion, that is much more time (and memory) consuming than merely analyzing dependences. For example, optimizing the computationally-intensive subroutine, *rhs* (containing 106 statements within 3 large loop nests), in the *applu* benchmark application from the SPEC OMP2012 Suite consumes nearly 3 hours using the PLuTo polyhedral compiler whereas the instance-wise dependence analysis takes less than a second. Similar is observed in multiple scientific applications that contain significant opportunity for data reuse through global program transformation. Thus, we conclude that the real problem of finding good global program transformations in a scalable (i.e. time- and memory-efficient) way has not been addressed by the compiler community.

Having evolved over the last two decades, polyhedral compilers, armed with exact dependence analysis and seamless transformation composition, have proven adept at performing global program transformation. This is especially useful because their traditional counterparts (including production compilers such as *gcc* and *icc*) are plagued by some of the artificial limitations such as non-conformable loop bounds, limitations of phase ordering and difficulty in composing transformations. This has restricted them to optimizing individual loops, or fuse consecutive loops with same loop bounds and loop order. However, while the polyhedral compilers have shown promise in effectively optimizing large scopes (89), the compile time and memory requirement is prohibitively expensive. The unscalability in polyhedral compilers stems from massive (fifth degree polynomial) increase in compile time with the number of statements (122). This unscalability of the employed techniques in polyhedral compilers has thus been recognized as a key problem in the community.

In this work, we address the scalability problem within a polyhedral compiler[1]. The essential ideas of our technique can be understood by an example program, shown in Figure 7.1a. The example program is constructed to represent some of the characteristic features in application programs. These features include updating the value arrays such as $rsd1$ and $rsd2$, and also frequent use of temporary variables such as $flux1$ and $flux2$ as shown in the figure.

The first important insight that leads to our proposed strategy is that statements within the same Strongly Connected Component (SCC) at a given loop-level undergo the same transformation at that level. Hence, statements within an SCC at the innermost loop-level undergo the exact same transformation in the transformed program at all loop-levels. For example, the statements S2 and S4 belong to the same SCC at loop-level j of the nest, and therefore have the same transformation until then, but differ later on. However, statements S1 and S3 belong to the same SCC at the innermost loop-level k, and thus remain perfectly fused at the innermost loop also. Thus, we call an SCC at the innermost loop-level as an 'Optimization atom (O-atom)' since it is smallest block of statements that becomes part of an optimization and all statements within this block are optimized similarly. We can therefore represent an O-atom with a single representative statement (S$_{rep}$), and force all other statements in the same atom to have the same transformation as S$_{rep}$. Thus, the entire program can be seen as a collection of O-atoms instead of statements. Furthermore, this coarsened granularity can be used to prune dependences. That is, dependences that have the same source and destination O-atoms, and the same dependence distance are clubbed into a single class with a single representative dependence D$_{rep}$. Since many dependences have small dependence distance and are therefore similar, there is significant opportuntity to reduce the effective number of dependences to be considered. Those dependences whose source (or destination) statement is not S$_{rep}$ but another statement in the same O-atom, are artificially assigned S$_{rep}$ as their source (or destination) in their respective O-atoms. This ensures that each S$_{rep}$ accounts for the constraints needed for correct transformation, while at the same time allows for the same to be achieved through fewer program statements and dependences.

Thus, the discovery of O-atoms in a program can lead to significant reduction in compile times and the memory requirement. However, it is not possible to find O-atoms in a given program. This is because, finding O-atoms rests on information about SCCs at the innermost loop-level. But, the innermost loop is unknown until the actual transformation is computed, since

[1]The techniques presented can, however, be implemented within traditional compilers as well

the transformation process may well involve loop interchange resulting in a different innermost loop than the original program. We thus propose to use 'Optimization molecules (O-molecule)' instead. An O-molecule is the block of statements comprising the loop body within any loop in the program. For example, statements S1-S3, S4-S5, S6-S8, and S9-S10 in Figure 7.1a form the O-molecules of the program. We call it an O-molecule because an O-molecule may be a collection of O-atoms because a loop body may contain multiple SCCs, just as statements S1 and S3 belong to the same SCC at the innermost loop k, while S2 is another SCC by itself at that level. Thus, O-molecules, unlike O-atoms, are easily recognizable even before program transformation begins. Also, since these are coarser than O-atoms, it allows for more pronounced reduction in analysis complexity. However, since all statements in an O-molecule are now forced to have the same transformation, optimizations such as loop distribution (used to enable vectorization) may be constrained. However, this does not hurt performance of the transformed program since backend compilers are adept at recognizing vectorization opportunities through distribution if needed.

This notion of 'O-molecule' becomes very useful in reducing the complexity of analysis. As a result of breaking our example program into *O-molecules*, the effective number of statements analyzed are just 4, instead of 10 in the original program as shown in Figure 7.1b. Figure 7.1b also shows that the reads and writes of all statements in an O-molecule are condensed onto the representative statement, S_{rep} within each O-molecule. This ensures that S_{rep} accounts for all the constraints needed to express a legal transformation. Moreover, the actual number of dependences analyzed by our framework are reduced from 64 to just 18. The reduction in the effective number of statements analyzed by our framework are more pronounced in case of real application programs that contain many more statements in a single loop. Also, the reduction in the number of dependences analyzed is a function of the number of different value arrays used in the program. This is because, value arrays such as $rsd1$ and $rsd2$ used in our example program tend to be updated in multiple loop nests in real applications that use many of them. Since they also have short dependence distances, this creates an opportunity to club many dependences into one class. For example, in the most computationally intensive subroutine, *rhs*, in the *lu* application program from the NPB/SPEC OMP2012 Benchmark Suite that contains 106 statements, 3033 dependences, and 30 value arrays reused in 3 large loop nests, the number of statements and dependences analyzed by our framework are reduced to 23 and 849, respectively. As a consequence of the reduction, the compile time and memory requirement are reduced by factors of 734 and 60, respectively.

The rest of the chapter is organized as follows. Section 7.2 discussed the precise causes of the unscalability of polyhedral compilers with respect to large programs. This is followed by a detailed discussion of the implementation of our technique within the polyhedral framework in Section 7.3. Section 7.4 demonstrates the performance of our technique with regards to compile time and memory requirement when compared to the state-of-the-art polyhedral compilers, and also the Intel production compiler. The related work is discussed in Section 7.5, and Section 7.6 concludes this work.

7.2 Causes of Unscalability

The bulk of the time spent in compilation through a polyhedral compiler is in 3 distinct phases. These are, (1) constructing the set of constraints in the coefficients of the transformation matrix (of each statement) from dependence polyhedra, (2) solving these linear constraints using an Integer Linear Programming (ILP) solver to obtain transformation matrix for each program statement, and (3) incorporating further constraints to ensure linear independence of the next solution (a legal hyperplane) with the previously found solutions. In this section, we describe how each of these phases proves to be a culprit in the unscalability of polyhedral compilers.

7.2.1 First Culprit: Phase I - Constructing legality constraints

As noted in Chapter 2, the linearized legality condition in Equation 2.6 must be satisfied for every dependence in the SCoP. Thus, the number of constraints increase linearly in the number of dependences, which themselves increase quadratically in the number of statements. This leads to quadratic increase in the number of constraints in program statements. Consequently, both the compile time and memory requirement increase significantly.

Increase in compile time. As discussed in Chapter 2, the Fourier Motzkin Elimination (FME) method is used to eliminate the Farkas multipliers when linearizing the legality condition. Each of the multipliers are eliminated one at a time. Running an elimination step over n inequalities results in an increase in the number of inequalities (to a maximum of $n^2/4$ inequalities (123)) for subsequent steps. The number of these steps are proportional to the number of statements, and thus in a large program, a considerable time is spent to eliminate all Farkas multipliers. Furthermore, this step of eliminating Farkas multipliers for linearizing legality condition is performed for every dependence and thus the overall contribution of Phase-I to the overall compile time is notable.

Increase in memory requirement. Phase-I is the only memory consuming phase in the entire compilation. The memory is used to hold the constraints contributed by each dependence edge. These constraints are input to the ILP solver to find hyperplane solutions. Since a dependence edge can enforce constraints on the transformation coefficients of any two program statements, the total number of variables used to express all the constraints (from all dependences) in a matrix (we call it the Constraint Matrix, C) equal $dim.|V|$, where dim denotes statement dimensionality. Thus, the width of the Constraint Matrix, C, is $dim.|V|$. The height of the matrix is given by $k.|E|$ since each dependence in E contributes a fixed number of constraints that are proportional to the sum of dimensionalities of source and dependence statements. Thanks to elimination of all Farkas multiplieras through Fourier Motzkin Elimination, the size of k is not very large. However still, the overall memory requirement becomes $dim^2.|V||E|$, or $|V|^3$. We find that for programs that contain around 100 statements, the overall memory requirement is already of the order of a few gigabytes.

7.2.2 Second Culprit: Phase II - Using an ILP solver to find legal hyperplanes

As mentioned above, the number of constraints (inequalities) input to the ILP solver equal $k.|E|$, and the number of variables involved are $dim.|V|$. We thus estimate the (time) complexity of the ILP solver as follows. Let $Z(m, n)$ be the complexity of ILP using the Simplex algorithm with m constraints and n variables. Then, on a typical input, $Z(m, n)$ = $\mathbf{O}((m+n)mn)$ on average (without counting the bit-size complexity). In our ILP, this amounts to a complexity of $(|E|+|V|)|E||V|$, or $|E|^2|V|$, or $|V|^5$, i.e. a quintic complexity in the number of statements. This leads to large compilation times for SCoPs with many statements.

7.2.3 Third Culprit: Phase III - Finding linearly independent hyperplanes

Using an ILP solver, we find solutions to statement-wise loop hyperplanes one at a time, starting with the hyperplane corresponding to the outermost loop. However, it is required that we find as many independent hyperplanes as the dimensionality of the polytope (i.e the maximum depth of any statement within the SCoP). Thus, before a hyperplane for the next loop-level is found, it is essential to augment the ILP formulation with additional constraints that ensure that the next hyperplane found is linearly independent to those previously found. For this purpose, a sub-space ($H_S^{\perp}$) orthogonal to H_S (where H_S represents the hyperplane solutions found so far), is constructed as follows:

$$H_S^{\perp} = I - H_S^T (H_S H_S^T)^{-1} \, H_S \tag{7.1}$$

Clearly, $H_S^{\perp}.H_S = I$. For linear independence of the next hyperplane **h'** with all hyperplanes in H_S, the following must hold:

$$H_S^{\perp}.\mathbf{h'} \neq \mathbf{0} \tag{7.2}$$

Furthermore, these additional constraints need to be constructed in such a way that when combined with the existing legality constraints, a solution of the coefficients of the next hyperplane is still possible. This involves computing the intersection of the integer sets given by Equation 7.2 and the constraints in the coefficients of hyperplanes obtained by eliminating all Farkas multipliers in Equation 2.6. This set intersection computed using the Integer Set Library (ISL) (124) further involves simplication of the constraints over many steps until the actual intersection is computed. The number of these simplification steps are proportional to the number of dependences, $|E|$. Further, each step has a complexity of $\mathbf{O}(|E|)$. Since the hyerplanes found are statement-wise, the linear independence constraints are also statement-wise, and thus, the complexity of this step becomes, $\mathbf{O}(|E|^2.|V|)$, or $\mathbf{O}(|V|^5)$, i.e. also quintic in the number of statements as for Phase II. However, unlike Phase II, the quintic complexity of Phase III is more realistic, and we actually find this step to be most time consuming for large programs.

Thus, we find that among all three culprits, the common motive behind the unscalability problem is the large number of statements within a single SCoP, which also amounts to large number of dependences. One solution to the problem could be to break a single large SCoP into multiple smaller SCoPs, but that breaks the very purpose of achieving global program transformations such as fusing multiple loop nests for reuse. Thus, we propose our solution that solves unscalability without loosing any global optimization opportunities. *We attack the problem at the root, i.e. we effectively cut down the number of statements, and also the number of dependences in a program through our* ***Statement Condensation*** *and* ***Dependence Condensation*** *algorithms. The program is then represented with a smaller set of representative statements and dependences, which still allows to correctly reason about the application of any particular transformation. This cut in the number of statements (and dependences) serves as a one-shot* *solution to all the above causes of unscalability, and allows to compile SCoPs with many statements in affordable time and memory budget.* The following section details the algorithm used to implement our solution.

7.3 Implementation

As discussed in Section 7.1, *O-molecules* form our smallest participating units in the optimization process within polyhedral framework. That is, the granularity of polyhedral transformations is coarsened from individual statements. It was also discussed that this coarsened granularity opens up avenues for reducing the number of dependences and statements to be analyzed for transformation. Algorithm 5 details the algorithm that clubs dependences belonging to the same source and destination *O-molecules* and with the same dependence distance. A single dependence from each club (D_{rep}) is then sufficient to contribute to the dependence-wise legality constraints given by Equation 2.6.

Since the polyhedral framework computes statement-wise loop hyperplane, each legality constraint (pertaining to a dependence) is a linear function of the coefficients of the hyperplanes of both the source and destination statements of the dependence. And, it is quite likely that a D_{rep} has a different source or destination than a chosen S_{rep} in its source or destination *O-molecule*. For example, in the example program shown in Figure 7.2a, suppose Statements S1 and S5 are chosen to be the 2 representative statements for the 2 loops. Further, consider the dependences belonging to the club denoting a dependence distance of +1 in loop k - the dependences between statements S3 and S6 on $flux1$, and statements S4 and S7 on $flux2$, both belong to this club. Clearly, neither of their source or destination statements is S_{rep} in the respective *O-molecules*. In such a scenario, S_{rep} cannot factually represent the *O-molecule* because none of the dependences that actually emanate from S_{rep} have a dependence distance of +1. Failing to account for the constraints that arise from dependences belonging to the club with distance +1 such as that between statements S3 and S6 leads to incorrectly transformed code shown in Figure 7.2b - the code does not respect the dependences between statements S3 and S6, and also statements S4 and S7.

In order for S_{rep} to factually become the representative statement for an *O-molecule*, the constraints on its hyperplane coefficients should reflect the constraints on the hyperplane coefficients imposed by those D_{rep}s whose source or destination *O-molecule* is the same as that of the S_{rep} in concern. This ensures that the transformation effected by the hyperplane found

ALGORITHM 5: Dependence condensation

```
1:  INPUT:
    $deps$ : Copy of the list of dependences
    $dist[][]$ : Array that stores the list of information such as dependence distances for each
    D$_{rep}$ with a given source and destination O-molecule
2:  begin
3:  for each dependence $d \in deps$ do
4:     $head$ = dist[$d$.src_mol][$d$.dst_mol]
5:     if $head$ = $\emptyset$ then
6:        Allocate space at $head$
7:        $head$.info $\equiv$ d.info
8:        $head$.next = $\emptyset$
9:     else
10:       $tmp$ = $head$
11:       while $tmp$.next $\neq \emptyset$ do
12:          if d.info $\equiv$ $tmp$.info then
13:             Break out of while loop
14:          $tmp$ = $tmp$.next
15:       if $tmp$ = $\emptyset$ then
16:          Allocate space at $tmp$
17:          $tmp$.info $\equiv$ d.info
18:          $tmp$.next = $\emptyset$
19: end
20: OUTPUT: $dist[][]$ containing info about D$_{rep}$s
```

(using an ILP solver) for S$_{rep}$ is such that it satisfies all important dependences incurred by all statements in the *O-molecule*. In other words, if a dependence such as that between statements S3 and S6 precludes fusion of the loops (which would have been otherwise legal when only considering the dependences involving S$_{rep}$), then such a fusion will not be performed because the constraints imposed by that dependence have been transferred onto S$_{rep}$. As a result, S$_{rep}$ factually becomes a representative statement, and all other statements in the *O-molecule* can then safely assume the same transformation as that of S$_{rep}$.

Our framework achieves this reduction in effective number of program statements by first choosing any one statement from each *O-molecule* as S$_{rep}$. We particularly choose that statement in any *O-molecule* that becomes the source of first such dependence found in a straightforward traversal of the list of D$_{rep}$s. Algorithm 6 shows the matrix, cst that accumulates the

```
for(i=0;i<N;i++) {
 for(j=0;j<N;j++) {

  for(k=0;k<N;k++) {
   S1: rsd3[i][j][k] = c2 * rho[i][j][k];
   S2: c1 = 0.5 * rho[i][j][k];
   S3: flux1[k] = c2 * u[i][j][k];
   S4: flux2[k] = c1 * u[i][j][k]; }

  for(k=1;k<N-1;k++) {
   S5: rsd3[i][j][k] = c3 * c4 * rsd3[i][j][k];
   S6: rsd1[i][j][k] = rsd1[i][j][k] - c3 *
                       (flux1[k+1] - flux1[k-1]);
   S7: rsd2[i][j][k] = rsd2[i][j][k] - c4 *
                       (flux2[k+1] - flux2[k-1]); }

}}
```

(a)

```
for (i=0;i<=N-1;i++) {
 for (j=0;j<=N-1;j++) {

  c1=0.5*rho[i][j][0];
  flux1[0]=c2*u[i][j][0];
  flux2[0]=c1*u[i][j][0];
  rsd3[i][j][0]=c2*rho[i][j][0];

  for (k=1;k<=N-2;k++) {
   c1=0.5*rho[i][j][k];
   flux1[k]=c2*u[i][j][k];
   flux2[k]=c1*u[i][j][k];
   rsd3[i][j][k]=c2*rho[i][j][k];
   rsd3[i][j][k]=c3*c4*rsd3[i][j][k];
   rsd1[i][j][k]=rsd1[i][j][k] - c3 *
                 (flux1[k+1]-flux1[k-1]);
   rsd2[i][j][k]=rsd2[i][j][k] - c4 *
                 (flux2[k+1]-flux2[k-1]); }

  c1=0.5*rho[i][j][N-1];
  flux1[N-1]=c2*u[i][j][N-1];
  flux2[N-1]=c1*u[i][j][N-1];
  rsd3[i][j][N-1]=c2*rho[i][j][N-1];

}}
```

(b)

```
for (i=0;i<=N-1;i++) {
 for (j=0;j<=N-1;j++) {

  for (k=0;k<=1;k++) {
   c1=0.5*rho[i][j][k];
   flux1[k]=c2*u[i][j][k];
   flux2[k]=c1*u[i][j][k];
   rsd3[i][j][k]=c2*rho[i][j][k]; }

  for (k=2;k<=N-1;k++) {
   c1=0.5*rho[i][j][k];
   flux1[k]=c2*u[i][j][k];
   flux2[k]=c1*u[i][j][k];
   rsd3[i][j][k]=c2*rho[i][j][k];
   rsd3[i][j][k-1]=c3*c4*rsd3[i][j][k-1];
   rsd1[i][j][k-1]=rsd1[i][j][k-1] - c3 *
                   (flux1[k]-flux1[k-2]);
   rsd2[i][j][k-1]=rsd2[i][j][k-1] - c4 *
                   (flux2[k]-flux2[k-2]); }

}}
```

(c)

Figure 7.2: (a) Original program; (b) Incorrectly transformed (fused) program; (c) Correctly Transformed program (with shifting in S5-S7)

dependence-wise legality constraints. Columns of cst correspond to the input variables of the ILP, i.e. the coefficients of the hyperplane of each statement in the program. Originally, each dependence would add constraints in cst. Using Algorithm 5, we cut down those dependences to the set D$_{rep}$. Now, further, each dependence in D$_{rep}$ adds constraints to only the variables representing coefficients of S$_{rep}$ as shown in Algorithm 6. This reduces the number of variables that are input to the ILP solver from being a function of the statements in the program, to the number of *O-molecules*, which is a significant factor of reduction.

7.4 Experimental Evaluation

7.4.1 Setup

The test programs were compiled (and run) on an Intel Xeon processor (E5-2650) with 8 Sandy Bridge-EP cores, operating at 2.0GHz. The processor has private L1 (32KB per core) and L2 (256KB per core) caches and a 20MB shared L3 cache, and 8GB memory. In this section, we compare the time and memory consumed for compilation by the state-of-the-art PLuTo polyhedral compiler and our work on statement and dependence condensation, as presented in this chapter. In addition, we also show the time and memory consumed for compilation by the state-of-the-art Intel production compiler. The Intel compiler, although very efficient in terms of time and memory consumption for compilation, is very conservative in analyzing large scopes for global program transformation. The reader is therefore referred back to the performance results in Chapter 5 when using our framework, versus the Intel compiler.

ALGORITHM 6: Statement condensation

```
1: INPUT:
   $D_{rep}$ : Set of representative dependences constructed from dist[][]
2: Step 1: Finding set of representative statements, $S_{rep}$
3: begin
4: for each O-molecule $m$ in the program do
5:    for each dependence $d \in D_{rep}$ do
6:       if $d$.src_mol = $m$ then
7:          $S_{rep}$[m] = $d$.src_stmt
8: end
9: Step 2: Using $S_{rep}$[] to reduce the number of variables
10: begin
   {/* This step only shows program segments where $S_{rep}$[] constructed in Step 1 becomes
   useful */}
   {/* src_offset = f($d$.src_stmt)
   dest_offset = f($d$.dest_stmt) */}
11: src_offset = f($S_{rep}$[$d$.src_mol])
12: dest_offset = f($S_{rep}$[$d$.dest_mol])
13: for each coefficient $v$ in the source and destination statement's hyperplane do
14:    ...
15:    $cst$[f($d$)][src_offset+$v$] = ...
16:    $cst$[f($d$)][dest_offset+$v$] = ...
17:    ...
```

7.4.2 Benchmarks

The benchmarks used in these experiments are the same as used to show the results of our proposed *variable liberalization* optimization in Chapter 5. The benchmarks are listed in Table 7.1.

Benchmark	Benchmark Suite	Category	Problem Size
applu	NPB/OMP2012	Computational Fluid Dynamics (CFD)	N=102; CLASS B
bt	NPB/OMP2012	”	”
sp	NPB	”	”
zeusmp	CPU2006	Simulation of astrophysical phenomena	Reference Input

Table 7.1: Summary of the benchmarks

7.4.3 Results and Discussion

Table 7.2 compares the time and memory consumed for compilation by different compilers. It is important to note that the maximum memory required for any benchmark is around 6GB while the memory on the test processor is 8GB. This ensures that the compile times shown in our results are not influenced by frequent page faults that result when the memory used reaches the memory limit on the host. The table also shows the number of statements and the number of dependences for each of the 8 SCoPs chosen for our experiments. As mentioned in Chapter 5, performance results on such large SCoPs have not been shown using existing polyhedral compilers. In that chapter, we show that our framework that implements *variable liberalization* achieves an average (geometric average) speedup of 2.1x and 5.16x over the Intel Fortran compiler and the PLuTo polyhedral compiler, respectively. We also recognized in that chapter that the compile time and memory requirement of the polyhedral compilers is still prohibitively large for them to be employed for compilation in the real world - the benchmarks with around 100 statements take more than 3 hours to compile. Through our proposed technique of *Statement Condensation* and *Dependence condensation*, we are able to reduce the compilation time very significantly, with the largest compile time being merely 18 seconds for the *rhs* subroutine in the *lu* benchmark. We refer to our work (in the form of the 2 algorithms presented in this chapter to achieve scalability) as ***scalefuse***, which is implemented as another compiler option within the PLuTo compiler. On average (geometric average), *scalefuse* receives an improvement of 395x and 28x in compile time and memory requirement, respectively, over the PLuTo compiler. Also, the compile time of *scalefuse* is now comparable and the memory requirement is even less than the Intel compiler. These results clearly present a strong case for the practical use of a polyhedral compiler for optimizing application programs, in addition to kernels.

The last two columns in Table 7.2 show the number of representative statements and dependences actually analyzed by *scalefuse*. On average, the number of statements and dependences analyzed reduce by factors 4.8x and 7x, respectively. Since the time and memory complexity of the three key phases of compilation is polynomial with a large power in the number of statements, the gains from our proposed *statement condensation* (and *dependence condensation*) are large. In particular, the improvement in compile time in more pronounced since the compile time varies as $|V|^5$ while the memory requirement varies as $|V|^3$. Another interesting point to note is that the number of representative dependences (D_{rep}s) are larger in case of *bt*, *sp*, and *lu* benchmarks than the others because these benchmarks involve a lot more variables, while

Benchmark	Subroutine	# statements	# deps	dim	Compile time (s)/Memory req. (MB)			# S_{rep}	# D_{rep}
					icc	smartfuse	scalefuse		
bt	rhs	48	1419	4	4/116	303/703	4.11/61.88	15	391
sp	rhs	50	1428	4	4/116	360/766	4.132/77.93	15	391
lu	rhs	106	3033	4	1.9/71.7	11302/5542	15.39/92.22	23	849
zeusmp	hsmoc.1	121	2660	3	5/141	4903.72/1207.8	11.76/40.12	21	217
	hsmoc.2	118	2493	3	5.9/151	6784.99/1256.26	11.74/41.28	18	198
	hsmoc.3	120	2296	3	5.7/143	8121.38/1643.15	13.87/55.59	18	234
	lorentz.1	98	2227	3	3.1/95.2	6550.48/2548.09	6.51/62.44	21	256
	lorentz.2	92	2149	3	2.49/86	7759.11/3389.42	8.7/67.61	18	247

Table 7.2: Compile time, memory requirement of different compilers

Benchmark	Subroutine	Compile time (s)					
		Phase - I		Phase - II		Phase - III	
		smartfuse	scalefuse	smartfuse	scalefuse	smartfuse	scalefuse
bt	rhs	8.45	2.32	285.82	1.142	3.96	0.152
sp	rhs	9.22	2.334	340.32	1.144	4.58	0.157
lu	rhs	265.7	5.102	10822.91	8.316	111.83	0.577
zeusmp	hsmoc.1	213.7	8.317	4603.32	0.443	81.72	0.118
	hsmoc.2	182.5	6.69	6522.88	2.169	73.86	0.082
	hsmoc.3	160.58	8.41	7878.65	2.46	76.09	0.133
	lorentz.1	625.63	3.047	5780.06	1.226	141.6	0.177
	lorentz.2	904.10	4.26	6523.33	1.83	326.68	0.41

Table 7.3: Compile times for the three time consuming phases in polyhedral compilation

the subroutines in the *zeusmp* benchmark use a lot of temporary variables with the same name, revealing more opportunity for dependence condensation.

Table 7.3[2] shows the time taken by each of the three phases that we identify as culprits for long compile times in Section 7.2. Clearly, *scalefuse* outperforms PLuTo considerably in all three phases. As discussed in Section 7.2, Phase II is the most time consuming step in compilation. Consequently, the most notable reduction through *scalefuse* is also for Phase II, i.e. the phase where PLuTo adds constraints to guarantee linear independence of future hyperplanes with those already found.

7.5 Related Work

The problem of unscalability of compilers has been well recognized in the literature. Much of the past work has been devoted to speeding up either the dependence testing through newer

[2]The compile times and memory req. numbers shown in this table are different from those shown in Chapter 5 because we use the most recent version of the PLuTo compiler for this study, which enforces stricter constraints to obtain a feasible solution

(faster, and less conservative) dependence tests, or the construction of the Program Dependence Graph (PDG) to be used later in program optimization. However, these works have not translated into practical implementations in existing production compilers due to their weakness in both, (1) performing exact dependence analysis, and (2) composing effective program transformations. As a result, current production compilers fare poorly with regards to global program optimizations due to limiting themselves to small scopes. It is only more recently in the context of the more powerful optimizing compilers using the polytope model that the problem of scalability when applying global program optimizations to large program scopes, has come to the fore.

Dependence Testing. Dependence testing answers the question whether two statements (or statement instances) refer to the same memory location across iterations. Most dependence tests are approximate, yet conservative, i.e if a dependence exists, it is reported; however, false dependences may also be reported leading to (unnecessarily) constrained optimization. These include the fast GCD-test (125), the Banerjee test (126), the Power test (127), etc. There are also exact tests that give an exact solution, but at a higher computational cost, such as the Omega-test (128), and the PIP-test (129). *Instance-wise dependence analysis* (80) subsumes previous work in dependence testing for it precisely tells which instances of the involved statements are in dependence. Thus, this instance-wise analysis is the most powerful *dependence abstraction* than its less precise counterparts used in previously proposed testing algorithms, such as (1) *dependence levels* (130) that tells which loops in the loop-nest carry a dependence, and (2) *distance vectors* (131; 132) that shows the difference of the loop counters of dependent instances. As a result, it aids the application of finer program transformations such as those in the polyhedral compilers.

Scalable PDG construction. In order to speed up the construction of PDG while not sacrificing precision, Maydan et al. (133) propose to use a collection of different dependence tests. They show that faster tests such as the GCD-test suffice for most dependences, while expensive tests can be used for remaining dependences. They also use memoization to avoid initiating a test on those dependences which are similar to those seen already. Recently, (70) propose a method to enable fast and precise construction of the Directed Acyclic Graph of Strongly Connected Components (DAG$_{SCC}$, also called condensation of the PDG). The authors recognize the use of the DAG$_{SCC}$ in program optimization rather than the PDG, and thus propose to directly compute the former by analyzing fewer dependences. This approach, although more closely tied to program transformation, still does not address the real problem of speeding (scaling) up the step of finding global program transformations from the computed DAG$_{SCC}$. This, as discussed in Section 7.1 is the most time consuming step in compilation. In addition, the authors analyze individual hot loops, and do not take advantage of global program transformations within the polyhedral framework.

Scalability within Polytope model. The problem of unscalability of polyhedral techniques (the simplex algorithm for solving the linear programs for scheduling) when applied to large scopes was first recognized by Feautrier in (134). He also proposed to avert the problem by calling the simplex algorithm on smaller scopes or *modules* that are comparable to functions. Each of these modules could be optimized in much less time than all of them together. However, this approach misses the essential global optimizations such as fusing those modules and then parallelizing the fused structure. Upadrasta and Cohen (122) have identified this problem and have proposed a sub-polyhedral technique using (Unit-)Two-Variable-Per-Inequality or (U)TVPI Polyhedra. Their technique is an under-approximation of a general polyhedron, and thus does not guarantee to find best solutions. If the under-approximation results in an empty polyhedra (or a non-solution), the authors propose to distribute loops to achieve a solution. This will again introduce sub-optimal solutions. However, the authors show the scalability of their technique over the simplex algorithm used in PLuTo for highly unrolled *matmul* kernel. The *matmul* kernel although unrolled to introduce dependences on the order of thousands, still takes less than 5 seconds to compile using PLuTo, given the very simple nature of dependences. We believe that achieving a feasible solution by such an approach may be a challenge when using real application programs that contain imperfectly nested loops in multiple nests with different loop order. Our work in this chapter does not use any under-approximations, and uses knowledge of program semantics to cut-down the size of input (in terms of variables and constraints) to the simplex algorithm. We also demonstrate its scalability on real-world applications.

7.6 Conclusion

In this work, we address a key problem in the polyhedral (parallelizing) compilers: unscalability of the employed algorithms. We identify that the root cause of this problem is the large number of program statements (and program dependences) in the hot regions within real application programs. We therefore propose a one-shot solution to the problem of unscalability of the employed algorithms - represent the entire hot region with a much fewer semantically equivalent set of representative statements and dependences. Our algorithms for the purpose are based on the notion of ‘O-molecules’, i.e. a sequence of statements within innermost loops in the program, all of which are forced to undergo the same program transformation. This allows to achieve the needed statement (and dependence) condensation, and does not constrain program transformation significantly. As a result of our proposed *statement condensation* and *dependence condensation*, we can significantly reduce the compile time and memory requirement with respect to the state-of-the-art polyhedral compilers, and match those of the Intel compiler, while achieving an average execution performance improvement of 2.1x over the Intel compiler for the same 8 hot regions. Thus, the strengths of program analysis and transformation offered by a polyhedral compiler can now be effectively and practically employed to optimize real application programs in addition to kernels.

Chapter 8

Conclusion and Discussion

In this dissertation, we identify 3 key challenges faced by parallelizing compilers, particularly in applying optimizations to improve the critical memory performance of existing multi- and many-core processors. This is important because the hardware can assist in improving memory performance to a certain extent such as by providing caches, better prefetchers, etc, but it is extremely difficult for the hardware to excavate opportunities from the application code to make efficient use of these resources even if it is possible. It is certainly unrealistic to pass this responsibility to the programmer, because the host platforms tend to be so different and evolving, and NOT all programmers like to be 'close to the silicon' or feel comfortable at reasoning an optimization for long sequences of loop nests at their disposal. It is therefore, the most important responsibility of the compiler along with extracting parallelism to improve memory performance for achieving effective parallel performance.

8.1 Summary of important findings and directions for future research

In order to address the impact of important microarchitectural changes over the last decade on important memory optimizations such as loop tiling (for locality enhancement) and data prefetching (for latency hiding), we first of all study the nature of the interaction between hardware and compiler in such cases. Then, an algorithm is presented for each of these two optimizations in Chapters 3 and 4, respectively, that builds up on an understanding of the precise influence of hardware on these optimizations, and vice-versa. In particular, we find that loop tiling closely interacts with multi-level caches, their set-associativities, multithreading, vectorization, and hardware prefetching. Data prefetching, on the other hand, depends upon the hardware supporting prefetching in each host hardware and the type of interaction between hardware prefetching and compiler prefetching allowed by the host. In addition, prefetching also closely interacts with multithreading and vectorization. We find that incorporating these algorithms in the compiler gives significant performance improvement in both kernels and real applications, over state-of-the-art production compilers that largely ignore this important interaction between the compiler and host hardware. Also, since these algorithms are designed to take information about hardware's parameters as input, they can be deployed in a general-purpose compiler running on different hosts with their own specific parameters.

For future research, our initial findings indicate that many applications can benefit when loop tiling is performed with a precise understanding of the nature of hardware prefetching functional at multiple levels of cache in the modern hardware. For example, in an experiment, we found that the best tiled code was that where the tile size was simply chosen to fit the last level cache. Although, it was expected to not perform better than the tile size that fits the L1 cache, the reason for better performance stems from the fact that the hardware prefetcher itself brings the data to the L1 cache for faster accesses. Meanwhile, having larger tiles that better benefit from prefetching is more useful. This approach may also remove the high dependency on knowing the problem size of the code to implement the most effective tiling. Also, we believe that with an improvement in the memory system's performance through effective memory optimizations (such as loop tiling and prefetching), the memory hierarchies of modern multi-cores can be redesigned for better power efficiency. Particularly, we find that in some applications, it is not necessary to have 3-level cache hierarchy and 2-level caches are sufficient since prefetching can effectively bring the data to the L1 cache. With a closer study of the different range of target applications, a generic solution suitable for all applications can be found with a less power-consuming memory system.

In our research, we also focus on the loop fusion optimization, which is of particular relevance for achieving parallel performance in multi- and many-core processors. Ironically, state-of-the-art compilers are particularly ineffective in applying loop fusion. We identify the principal reason for this shortcoming to be their focus on optimizing small scopes such as individual loop nests. We address this problem by (1) proposing *variable liberalization*, an optimization that facilitates fusion of loop nests in the presence of transformation-restricting dependences on temporary variables, and (2) further, giving a heauristic-based cost model that wisely decides on the statements to be fused to achieve good reuse in the final transformed program without paying the expense of lost parallelism. We believe that the combination of these contributions presented, respectively, in Chapters 5 and 6, will allow production compilers to effectively fuse loop nests in real application programs, that have been shown to exhibit significant opportunity of temporal reuse through loop fusion.

Finally, Chapter 7 addresses the issue of compiler scalability, which is a longstanding issue that has kept the compilers from attempting major optimizations on large sequences of loop-nests as found in real applications. The compilers chose to avoid these attempts to contain the compile time and memory requirement for such compilation, as a programmer who has to wait for long to compile his code will shift to a different compiler. We introduce the notion of *Optimization-molecule* as the smallest grain of statement-block for which all dependences incurred can be subsumed in a single representative statement. This leads to a very significant reduction in constraints to be analyzed for reasoning transformations, and hence also compile times and memory requirement. This work, therefore, perfectly complements our work on loop fusion for large programs, and makes it an implementable solution for production compilers. We believe that this will also open up avenues for further research on global program optimizations, and is therefore an interesting direction for future research. Also, for the benefits of this work to become available to a larget set of applications, an extension to the polyhedral framework for non-affine codes will be extremely useful.

8.2 Final words

This thesis revisits important compiler optimizations for improving memory performance. These optimizations will find ever-increasing importance in future architectures, that are expected to face the challenges of the memory wall and bandwidth wall with increasing intensity. We believe that the most important general finding of this work is that the best solutions emerge when the hardware and software work in consonance for any given application. Since our work lies at the junction of architecture, compilers, and applications, it contains directions to guide the design of future architectures and their compilers - architectures and compilers built with an understanding of the capabilities and limitations of each other, and knowledge of the target applications, will achieve the best performance. We hope to use our experience and knowledge obtained from this research in the design of such architectures and compilers in the future.

References

[1] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. *Clock rate versus IPC: The end of the road for conventional microarchitectures*, volume 28. ACM, 2000.

[2] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[3] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 279–290, New York, NY, USA, 1995. ACM.

[4] Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 492–499, New York, NY, USA, 1999. ACM.

[5] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[7] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC'2008*, pages 132–146. Springer Berlin / Heidelberg.

[8] M. Wolfe. More iteration space tiling. In *SC '89*, pages 655–664. ACM, 1989.

[9] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, PLDI '91, pages 30–44, New York, NY, USA, 1991. ACM.

[10] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108 – 120, 1992.

[11] Amy W Lim, Shih-Wei Liao, and Monica S Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *ACM SIGPLAN Notices*, volume 36, pages 103–112. ACM, 2001.

[12] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[13] Monica D. Lam, Edward E. Rothberg, and Michael 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, pages 63–74, New York, NY, USA, 1991. ACM.

[14] Karim Esseghir. *Improving Data Locality for Caches*. PhD thesis, Rice University, September 1993.

[15] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: an analytical representation of cache misses. In *ICS '97*, pages 317–324. ACM, 1997.

[16] V. Sarkar. Automatic selection of high-order transformations in the ibm xl fortran compilers. *IBM Journal of Research and Development*, 41(3):233 –264, may 1997.

[17] V. Sarkar and N. Megiddo. In *ISPASS '2000*, pages 146 –153, 2000.

[18] Jun Shirako, Kamal Sharma, Naznin Fauzia, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan, and Vivek Sarkar. Analytical bounds for optimal tile size selection. In *ETAPS International Conference on Compiler Construction (CC'12)*, Tallinn, Estonia, March 2012. Springer Verlag.

[19] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *ICS '97*, pages 340–347. ACM, 1997.

[20] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27:3 – 35, 2001.

[21] Matteo Frigo. A fast fourier transform compiler. In *PLDI '99*, pages 169–180. ACM, 1999.

[22] Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. Spl: a language and compiler for dsp algorithms. In *PLDI '01*, pages 298–308. ACM, 2001.

[23] Sanket Tavarageri, Loius-Noel Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Dynamic selection of tile sizes. In *Proceedings of the 18th International Conference on High Performance Computing*, HiPC '11, Bengaluru, India, December 2011.

[24] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. *The Journal of Supercomputing*, 24:43–67, 2003.

[25] Tomofumi Yuki, Lakshminarayanan Renganarayanan, Sanjay Rajopadhye, Charles Anderson, Alexandre E. Eichenberger, and Kevin O'Brien. Automatic creation of tile size selection models. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 190–199, New York, NY, USA, 2010. ACM.

[26] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43:101–113, June 2008.

[27] Gabriel Rivera and Chau-Wen Tseng. A comparison of compiler tiling algorithms. In *Compiler Construction*, pages 168–182. Springer, 1999.

[28] P R Panda, Hiroshi Nakamura, Nikil Dutt, and Alexandru Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *Computers, IEEE Transactions on*, 48(2):142–149, 1999.

[29] Chung-hsing Hsu and Ulrich Kremer. A quantitative analysis of tile size selection algorithms. *The Journal of Supercomputing*, 27(3):279–294, 2004.

[30] K. Yotov, X. Li, G. Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358 –386, feb. 2005.

[31] David Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30(3):181–221, 2002.

[32] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Languages and Compilers for Parallel Computing*, volume 589 of *Lecture Notes in Computer Science*, pages 328–343. Springer Berlin / Heidelberg, 1992.

[33] O. Temam, E.D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *SC '93*, pages 410 – 419, nov. 1993.

[34] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04*, pages 7–16, France, September 2004.

[35] Keith Cooper and Jeffrey Sandoval. Portable Techniques to Find Effective Memory Hierarchy Parameters. Technical report, 2011.

[36] Polybench. Available at `http://www-roc.inria.fr/~pouchet/software/polybench/`.

[37] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In *PLDI '01*, pages 286–297. ACM, 2001.

[38] A. Tiwari, Chun Chen, J. Chame, M. Hall, and J.K. Hollingsworth. A scalable autotuning framework for compiler optimization. In *IPDPS '09*, pages 1 –12, may 2009.

[39] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. *U. of Southern California, Tech. Rep*, pages 08–897, 2008.

[40] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: towards automated performance tuning. In *SC '02*, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[41] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35(1):4:1–4:14, July 2008.

[42] Kaushik Datta. Auto-tuning Stencil Codes for Cache-Based Multicore Platforms. Technical report, University of California at Berkeley, Berkeley, December 2009.

[43] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO '05*, pages 111–122, 2005.

[44] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 200–209, New York, NY, USA, 2010. ACM.

[45] Hung Q Le, William J Starke, J Stephen Fields, Francis P O'Connell, Dung Q Nguyen, Bruce J Ronchetti, Wolfram M Sauer, Eric M Schwarz, and Michael T Vaden. Ibm power6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[46] Tien-Fu Chen and Jean-Loup Baer. A performance study of software and hardware data prefetching schemes. In *ISCA'94*, pages 223–232.

[47] Rafael H Saavedra and Daeyeon Park. Improving the effectiveness of software prefetching with adaptive executions. In *PACT'96*, pages 68–78.

[48] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *ISCA'03*, pages 388–398.

[49] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it does not, and why. *TACO'2012*, 9(1):29.

[50] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS'92*, pages 62–73.

[51] Daniel J. Quinlan, Markus Schordan, Bobby Philip, and Markus Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In *LCPC'01*, pages 383–394.

[52] Intel compiler. Available at `http://software.intel.com/en-us/intel-parallel-studio-xe`.

[53] Intel's VTune. Available at `www.intel.com/Software/Products`.

[54] Abdel-Hameed Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. The efficacy of software prefetching and locality optimizations on future memory systems. *JILP'2004*, 6(7).

[55] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. pages 40–52.

[56] Alexander C. Klaiber and Henry M. Levy. An architecture for software controlled data prefetching. In *ISCA'91*, pages 43–53.

[57] Vatsa Santhanam, Edward H Gornish, and Wei-Chung Hsu. Data prefetching on the hp pa-8000. In *ISCA'97*, pages 264–273.

[58] George C Caragea, Alexandros Tzannes, Fuat Keceli, Rajeev Barua, and Uzi Vishkin. Resource-aware compiler prefetching for many-cores. In *ISPDC'2010*, pages 133–140.

[59] Rakesh Krishnaiyer, Emre Kultursay, Pankaj Chawla, Serguei Preis, Anatoly Zvezdin, and Hideki Saito. Compiler-based data prefetching and streaming non-temporal store generation for intel xeon phi coprocessor. In *Workshop on Multithreaded Architectures and Applications*, 2013.

[60] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *ASPLOS'11*, pages 393–404.

[61] Dongkeun Kim and Donald Yeung. Design and evaluation of compiler algorithms for pre-execution. In *ASPLOS'02*, pages 159–170.

[62] Seung Woo Son, Mahmut Kandemir, Mustafa Karakoy, and Dhruva Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. In *PPOPP'09*, pages 209–218.

[63] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3-4):421–444, 1998.

[64] L. Rauchwerger and D.A Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. *Parallel and Distributed Systems, IEEE Transactions on*, 10(2):160–180, Feb 1999.

[65] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *JPDC*, 64(1):108 – 134, 2004.

[66] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 389–400. ACM, 2010.

[67] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 144–156. IEEE, 2007.

[68] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 90–100, New York, NY, USA, 2008. ACM.

[69] Sylvain Girbal, Nicolas Vasilache, Cdric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *IJPP*, 34:261–317, 2006.

[70] Nick P. Johnson, Taewook Oh, Ayal Zaks, and David I. August. Fast condensation of the program dependence graph. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 39–50, New York, NY, USA, 2013. ACM.

[71] Ken Kennedy and Kathryn McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *LCPC*, volume 768 of *Lecture Notes in Computer Science*, pages 301–320. 1994.

[72] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *SPAA*, pages 282–291. ACM, 1997.

[73] S. K. Singhai and K. S. McKinley. A parametrized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[74] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A unified framework for schedule and storage optimization. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 232–242, New York, NY, USA, 2001. ACM.

[75] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 53–65, New York, NY, USA, 1990. ACM.

[76] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3-4):649–671, 1998.

[77] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):773–815, 2000.

[78] Alain Darte and Guillaume Huard. New complexity results on array contraction and related problems. *Journal of VLSI signal processing systems for signal, image and video technology*, 40(1):35–55, 2005.

[79] Albert Cohen. Parallelization via constrained storage mapping optimization. In *High Performance Computing*, pages 83–94. Springer, 1999.

[80] Nicolas Vasilache, Cédric Bastoul, Albert Cohen, and Sylvain Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 335–344. ACM, 2006.

[81] Sebastian Hack, Daniel Grund, and Gerhard Goos. Register allocation for programs in ssa-form. In *Compiler Construction*, pages 247–262. Springer, 2006.

[82] Dror E Maydan, Saman P Amarasinghe, and Monica S Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 2–15. ACM, 1993.

[83] Riyadh Baghdadi, Albert Cohen, Sven Verdoolaege, and Konrad Trifunović. Improved loop tiling based on the removal of spurious false dependences. *ACM Trans. Archit. Code Optim.*, 9(4):52:1–52:26, January 2013.

[84] Konrad Trifunovic, Albert Cohen, Razya Ladelsky, and Feng Li. Elimination of memory-based dependences for loop-nest optimization and parallelization. In *3rd GCC Research Opportunities Workshop, GROW*, 2011.

[85] Nicolas Vasilache, Albert Cohen, and Louis-Noël Pouchet. Automatic correction of loop transformations. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 292–304. IEEE Computer Society, 2007.

[86] Nicolas Vasilache, Benoit Meister, Albert Hartono, Muthu Baskaran, David Wohlford, and Richard Lethin. Trading off memory for parallelism quality. In *International Workshop on Polyhedral Compilation Techniques, IMPACT*, 2012.

[87] Polyopt: a polyhedral optimizer for the rose compiler. Available at `http://www.cse.ohio-state.edu/~pouchet/software/polyopt/`.

[88] Rose compiler. Available at `http://rosecompiler.org/`.

[89] Sanyam Mehta, Pei-Hung Lin, and Pen-Chung Yew. Revisiting loop fusion in the polyhedral framework. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 233–246, New York, NY, USA, 2014. ACM.

[90] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proceedings of the 6th international conference on Supercomputing*, pages 313–322. ACM, 1992.

[91] Peng Tu and David Padua. Automatic array privatization. In *Languages and Compilers for Parallel Computing*, pages 500–521. Springer, 1994.

[92] Paul Feautrier. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing*, pages 429–441. ACM, 1988.

[93] Ken Kennedy and Kathryn McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *LCPC*, volume 768 of *Lecture Notes in Computer Science*, pages 301–320. 1994.

[94] S. K. Singhai and K. S. McKinley. A parametrized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

[95] Nimrod Megiddo and Vivek Sarkar. Optimal weighted loop fusion for parallel programs. In *SPAA '97*, pages 282–291. ACM.

[96] Chen Ding and Ken Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *JPDC*, 64(1):108 – 134, 2004.

[97] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: convexity, pruning and optimization. In *POPL '2011*, pages 549–562. ACM.

[98] Pluto: An automatic parallelizer and locality optimizer for multicores. Available at `http://pluto-compiler.sourceforge.net`.

[99] Pocc: Polyhedral compiler collection. Available at `http://www.cse.ohio-state.edu/~pouchet/software/pocc/`.

[100] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *IMPACT '2011*.

[101] Ibm xl compiler. Available at `http://ibm.com/software/awdtools/xlcpp/`.

[102] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *CGO '07*, pages 144 –156.

[103] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part ii, multidimensional time. In *PLDI '08*, pages 90–100. ACM.

[104] LLVM. Polly: Polyhedral optimizations for llvm. Available at `http://polly.llvm.org/`.

[105] Alain Darte. On the complexity of loop fusion. In *PACT '99*, Washington, DC, USA. IEEE Computer Society.

[106] Micha Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.

[107] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *CC'2008*, pages 132–146. Springer Berlin / Heidelberg.

[108] Uday Bondhugula. *Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model*. PhD thesis, Ohio State University, 2010.

[109] R-stream compiler. Available at `https://www.reservoir.com/?q=rstream`.

[110] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *CC '2010*, pages 283–303, Paphos, Cyprus, March. Springer Verlag.

[111] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *IJPP*, pages 313–347, 1992.

[112] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multi-dimensional time. *IJPP*, pages 389–420, 1992.

[113] Wayne Kelly and William Pugh. Minimizing communication while preserving parallelism. In *ICS '1995*, pages 52–60. ACM Press.

[114] Martin Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. PhD thesis, University of Passau, 2004.

[115] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24:445 – 475, 1998.

[116] Amy W Lim, Gerald I Cheong, and Monica S Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ICS '99*, pages 228–237. ACM.

[117] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *PACT '2010*, pages 343–352. ACM.

[118] Sylvain Girbal, Nicolas Vasilache, Cdric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *IJPP*, 34:261–317, 2006.

[119] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *ICS '05*, pages 151–160. ACM.

[120] Arvind J. Thadhani. Factors affecting programmer productivity during application development. *IBM Systems Journal*, 23(1):19–35, 1984.

[121] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[122] Ramakrishna Upadrasta and Albert Cohen. Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 483–496, New York, NY, USA, 2013. ACM.

[123] George B Dantzig and B Curtis Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory, Series A*, 14(3):288–297, 1973.

[124] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer Berlin Heidelberg, 2010.

[125] Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[126] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[127] Michael Wolfe and C-W Tseng. The power test for data dependence. *Parallel and Distributed Systems, IEEE Transactions on*, 3(5):591–601, 1992.

[128] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[129] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Op'erationnelle*, 22, 1988.

[130] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9:491–542, 1987.

[131] Leslie Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, February 1974.

[132] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[133] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 1–14, New York, NY, USA, 1991. ACM.

[134] Paul Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34(5):459–487, 2006.