Runtime Prediction of Fused Linear Algebra in a Compiler Framework

by

Ian Karlin

B.S., University of California at Davis, 2005

M.S., University of Colorado, Boulder, 2007

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2011

This thesis entitled:
Runtime Prediction of Fused Linear Algebra in a Compiler Framework
written by Ian Karlin
has been approved for the Department of Computer Science

Elizabeth Jessup

Prof. Jeremy Siek

Prof. Xiao-Chuan Cai

Prof. Manish Vachharajani

Dr. Jonathan Hu

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Karlin, Ian (Ph.D., Computer Science)

Runtime Prediction of Fused Linear Algebra in a Compiler Framework

Thesis directed by Prof. Elizabeth Jessup

On modern processors, data transfer exceeds floating-point operations as the predominant cost in many linear algebra computations. For these memory-bound calculations, reducing data movement is often the only way to significantly increase their speed. One tuning technique that focuses on reducing memory accesses is loop fusion. However, determining the optimum amount of loop fusion to apply to a routine is difficult as fusion can both positively and negatively impact memory traffic.

In this thesis, we perform an in depth analysis of how loop fusion affects data movement throughout the memory hierarchy. The results of this analysis are used to create a memory model for fused linear algebra calculations. The model predicts data movement throughout the memory hierarchy. Included in its design are runtime and accuracy tradeoffs based on our fusion research. The model's memory traffic predictions are converted to runtime estimates that can be used to compare loop fusion variants on serial and shared memory parallel machines. We integrate our model into a compiler where its predictions often reduce compile times by 99% or more. The kernel produced by the compiler with the model turned on are usually the same as the optimal kernel for the target architecture found by exhaustively testing all possible loop fusion combinations.

Dedication

I dedicate this thesis to all the friends and family who helped me through the ups and downs of life and contributed to me reaching this point.

Acknowledgements

I would like to thank the National Science Foundation for funding the grants that made it possible for me to complete this research. I also thank Professor Jessup for the guidance and support she provided especially when it came to transforming the quality of my writing. I would like to thank and acknowledge my thesis committee for their time and help. In particular, Professor Siek, who wrote the initial version of the Build to Order (BTO) compiler. Additional thanks go to my other co-authors Geoffrey Belter, Erik Silkensen, Pavel Zelinsky and Thomas Nelson for help both in writing papers and ideas contributed during various discussions. In particular I would like to recognize Geoffrey whom I worked closely with during the integration of the BTO compiler with my model and the analysis of the two working together.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

The execution of mathematical kernels is often the most time consuming part of scientific applications in the diverse fields of atmospheric science [107], quantum physics [97], and structural engineering [95]. Reduction in the runtime of these mathematical kernels is achieved using various methods. The increasing speed and parallelism of modern computer hardware produces large reductions in the overall program runtime [106]. Algorithmic improvements, such as iterative methods, diminish runtimes by decreasing the number of operations that need to be performed [68]. Finally, software tuning techniques reduce kernel runtimes by improving the computation's speed [13, 30, 55]. For many mathematical kernels, such as matrix-matrix multiplication, the result is near optimal performance on a wide variety of hardware [49, 105]. The performance of these kernels also improves in lockstep with processor improvements.

Despite the large number of ways to raise performance, the speed of many calculations increases more slowly than processors and performance tuning technique advancements improve other operations. Most of these calculations perform few floating-point operations per memory access. The result is that their performance is bounded by the speed of the memory bus, which is slower than the CPU [27]. This speed difference is a consequence of memory bandwidth's increasing by 7-10% per year while the CPU's floating point throughput has increased by approximately 60% per year [52]. This trend has occurred for the past 30 years and is expected to continue. Therefore the speed of memory bound calculations increases at the slower rate of memory bandwidth improvements [3].

The combination of memory-bound algorithms and slower memory bandwidth increases means that many computations perform significantly below theoretical peak machine performance [21, 93]. To counteract this problem, work has been performed to reduce the amount of memory traffic required to perform calculations [3, 5, 10, 30, 55]. Loop fusion [44], an optimization that combines multiple loops together, is one such technique.

Fusing loops that access the same data elements can reduce the amount of data that must be read from memory. An added benefit of loop fusion is that it often combines operations in a way that makes array contraction possible [44]. Array contraction is an optimization that reduces the size of an array that is neither an input or output to a smaller array or scalar quantity. Array contraction reduces the working set size of a program and may further decrease data movement by diminishing the number of conflict misses in a cache. The result is fused codes that can run much faster than their unfused equivalents [10, 55, 101].

One frequent application of loop fusion is to memory bound linear algebra operations. Creating an efficient fused routine is important, since the kernel that can be fused often contributes a large portion of the overall routine runtime [55]. However, generating a routine with the optimal amount of fusion is not always easy or obvious because fusion interacts in both positive and negative ways with the memory hierarchy and other optimizations [61]. Without exploring the whole fusion search space, it is possible to produce a sub-optimal routine. However, the number of possible ways to create a fused routine is NP-complete [26], and, therefore it is not always feasible to enumerate and test all possible combinations.

Multiple approaches to creating efficient routines have been used by researchers, including optimizing single routines to run as fast as possible [55,101], performing fusion on a general purpose language input [16, 85], using a domain specific language and compiler [10] and adding important and commonly used functions to a widely used application programing interface [15]. The projects that accept diverse inputs often include cost models or heuristics to predict the performance of routines [10, 16, 86]. These models and heuristics decrease the time the compilers take to generate fused routines, though they sometimes produce routines with sub-optimal amounts of fusion [82].

The result of these efforts is highly efficient routines that can be over 100% faster than non-fused implementations.

In this thesis, we present a model designed with enough speed to handle large search spaces and enough accuracy to result in a feasible number of routines that must be tested to assure good performance [10, 65]. We integrate our model into the Build to Order (BTO) compiler framework [10] and demonstrate that it reduces the search space of compilation without significantly impacting routine quality. BTO is is a compiler that takes in an annotated subset of MATLAB and produces optimized kernels in C. Optimizations included within BTO include two forms of loop fusion and data partitioning. The data partitioning enables cache blocking and the creation of shared memory parallel codes [11]. BTO enumerates the entire search space of potentially profitable loop fusion combinations that access common data to avoid generating sub-optimal fused kernels, which is feasible for the restricted domain in which BTO works. Those operations are then tested using a hybrid analytical/empirical testing methodology. In this methodology, the model presented in this thesis is used to produce kernel runtime estimates and the best identified kernels are then empirically tested. The compiler then outputs the routine identified as the fastest. The result is efficient linear algebra kernels produced in a small amount of time that run over 100% faster than vendor-optimized Basic Linear Algebra Subprograms (BLAS) on serial and parallel machines.

In designing the model, we draw inspiration from the fact that, for memory bound operations, the best way to compare the performance of two implementations is by comparing their memory costs, not their operation counts [3]. Reducing the memory traffic of an operation results in a corresponding reduction in runtime of memory bound operations [30]. Additionally, the performance of memory-bound codes can be predicted from memory access patterns and benchmarks of the system [20]. Finally, we use only the most distinguishing memory effects for fused calculations to reduce model runtime. The result is a model that in a small amount of time accurately distinguishes between multiple versions of the same routine to find a small subset that can feasibly be empirically tested. Included in this thesis are models for both single processor and multiprocessor shared memory systems.

The rest of the thesis is organized as follows: In Chapter 2, we present an overview of the important parts of computer architecture as relevant to the performance of loop fused operations. We review performance tuning techniques other than loop fusion and their interactions with the computer architecture. Additionally, we describe other memory models and runtime prediction methods.

In Chapter 3, we review others' loop fusion work including the analysis of its complexity. Different ways to produce fused routines are presented in detail. The impact of fusion on the memory subsystem is explored, along with a detailed explanation of when and why fusion impacts data movement. We show ways to mitigate the negative effects of fusion while not losing positive effects. We also discuss the complexity of searching for the optimum amount of fusion and how to reduce decisions considered without missing the best performing routines. Included in this chapter is an overview of the BTO compiler system its capabilities and features and how we leverage it to test, develop and improve our model.

We explain how we model memory traffic in Chapter 4. We discuss the assumptions made in the model and the accuracy and performance tradeoffs that result. The chapter also contains details of the theoretical framework for the model, discussion of its implementation and analysis of the model's accuracy, strengths and weaknesses.

Chapter 5 describes how memory predictions are converted into runtime predictions. It covers the method used to perform this translation and the accuracy of these predictions. Included is a theoretical framework and implementation details. We also explain a procedure to automatically determine hardware characteristics of a machine relevant to memory and runtime prediction.

We expand our memory traffic runtime predictions to shared memory parallel machines in Chapter 6. Included in this expansion are runtime predictions for routines on parallel machines. Included in this chapter are assumptions made about parallel machines, differences between modeling serial and parallel machines and validation of the model's accuracy.

In Chapter 7 we describe how the model and runtime prediction function are integrated and used within the BTO compiler system. We show how and by how much the model reduces the

amount of time required to find an efficient kernel.

Finally, Chapter 8 includes conclusions, and Chapter 9 presents areas for future work.

Chapter 2

Background

To achieve good performance, programmers of linear algebra software must account for the underlying hardware on which their programs are run [49, 106]. Tuning of routines for the target architecture is important because the speed differential between naive and tuned algorithms is often significant [5,6,100,106] due to the naive algorithm's inefficient use of the computer's hardware [101]. Memory models [41,47] and automated tuning programs [13,42,98,105] aid in the creation of efficient programs. As computer architectures continue to improve and become increasingly complex, new models and tuning techniques are to needed aid the implementation of high performing routines on modern hardware [31].

In this chapter we begin with a review of the important elements of computer architecture for linear algebra operations for serial and parallel systems. We then present an analysis of various tuning techniques and their interaction with computer hardware. A summary of some of the major libraries that perform linear algebra computations follows. Next, we present a description of memory models used to aid in performance analysis and tuning of linear algebra computations. Then we discuss other approaches to autotuning software. Finally, we present how others incorporate models into hybrid search strategies. Throughout this chapter and the rest of this thesis linear algebra computations are presented. When they are, all variables that are capital Roman letters denote matrices, lower case Roman letters are vectors and lower case Greek letters are scalar quantities.

2.1 Computer Architecture

Creating high performing linear algebra kernels requires understanding the computer hardware on which the routines are run. For calculations that are bound by data movement, knowledge of the memory subsystem and how it impacts runtimes aids in tuning calculations. In this section, we describe in detail the memory subsystem and how data move through it. We also discuss the gap in data movement speed and the speed of the processor. Included in this section are the significant differences of the memory subsystem design for single processor and parallel machines.

2.1.1 Memory Subsystem

Efficient use of the memory subsystem is important to linear algebra routine performance [13,49,70,101]. To design efficient routines requires a good understanding of the memory subsystem, which is made up of multiple components or varying sizes and speeds. From the fastest and smallest to the largest and slowest there are: registers, caches and main memory. Ofter multiple caches of varying size and speed are used. These memory structures along with the translation lookaside buffer (TLB), which speeds up the process of finding where data is stored in computers running multiple processes, make up the memory hierarchy.

In this section, we explain in detail the components of a computer's memory hierarchy and how these structures combine to impact performance. The material is a summary of information covered in **Computer Architecture: A Quantitave Approach** by John Hennessy and David Patterson [52].

2.1.1.1 Registers

Within the processor, all operations and calculations occur by manipulating data that are stored in registers. Registers are high speed memory structures that typically store 32 or 64 bits of information that can be accessed nearly instantly by the processor. The speed of access comes at a large monetary and transistor cost. Therefore, most modern processor contain only 8 to 128 general

purpose registers available that are available for programs to use. Due to the limited number of registers most calculations require some or all of their data to be read from memory before the calculations begins.

2.1.1.2 Caches

To perform a calculation on data not within a register, a computer needs to retrieve data from its main memory and transfer it into the processor where the computation occurs. How long the transfer takes is dependent on the latency and bandwidth between main memory and the processor. Latency is the amount of time it takes to move data from memory to the processor, and bandwidth is the amount of data that can be moved between two parts of a computer system in a given time. They are the limiting factors in reading data from memory. Since main memory has a high latency and a low bandwidth to the processor, reading large amounts of data from main memory can limit the performance of a program. High latency affects the program's runtime by causing the processor to stall while waiting for data to be retrieved from main memory. A computer with low bandwidth cannot move data from main memory to the processor as rapidly as the processor can perform calculations. Therefore, decreasing latency and increasing bandwidth by reading as much data from closer faster parts of the memory hierarchy are important to decreasing a computer program's runtime.

The pattern and distance of reads from different memory addresses determines whether bandwidth or latency are more important to data retrieval speeds. When a program accesses main memory sequentially, bandwidth is more important than latency for performance. Once the initial memory location is accessed, data are streamed from consecutive memory locations as fast as the machine's bandwidth allows. When a program accesses data from nonsequential memory addresses, low latency is more important than high bandwidth for performance. Each access to a nonsequential memory address results in a high latency memory lookup, which can take a few hundred cycles.

To reduce the effects of high-latency and low bandwidth, caches are used. A cache is a fast

memory structure that stores a subset of the data in main memory. The data stored in a cache are organized into units called cache lines, which are typically 32 to 128 bytes in size. These lines contain consecutive memory addresses from main memory. The first cache line begins at the memory address 0, with the next line beginning immediately after the first line. Therefore, a cache line begins every x bytes, where x is the line size used by the cache. As a segment of data is read into the processor from main memory, it is put into the cache; at the same time, the other data members on the same cache line are also read into the cache.

There are two ways programs can take advantage of caches. Consecutively reading data from nearby memory address results in spatial locality of data accesses. For example spatially local reads occur when a program reads two data elements on the same cache line. When a program reads a data element that is already in cache due to a previous read the program is exhibiting good temporal locality. Each of these access patterns reduces the amount of data read from slower memory structures and speeds up routine execution.

Most caches are small in comparison to main memory. Typical cache sizes range from 32 KB to 24MB while main memory can be many gigabytes in size. Therefore, caches can only store a subset of main memory at a given time. To store and retrieve values from a cache quickly, the data are stored by using the first bits of the address in main memory to index into a set where typically two to sixteen cache lines are stored. Once the proper set is found, the next bits of the address are compared with the tag of the cache line to see if the set contains the line sought. If the line is found, a cache hit has occurred, and the data sought are read from the cache line to the processor. If the line that contains the data is not found then a cache miss has occurred. The cache line containing the data sought is then read from main memory and stored in the appropriate set. If the set is already full with other lines, then the least recently used line is evicted from the cache. The same procedure is followed for computers with multiple levels of cache. When there is a miss in a faster and smaller cache, the next larger and slower caches are tried. If the data are not found in any cache, they are read from main memory.

On most computers cache lines are arranged in sets to simplify cache design and reduce

manufacturing cost. For these set associative caches the number of lines in the cache in which a data element can be stored is the associativity of a cache. When a data element is read from memory it is stored in a cache set using a mapping based on where in memory it is stored. If that set is full the cache evicts the least recently used (LRU) cache line. Often the cost of maintaining which line was used last is too expensive or complicated and a pseudo-LRU replacement strategy is used.

However, some computers use fully associative caches where a data element can be stored in any cache line. In a fully associative cache all cache lines need to be checked for that line, whenever data is read. To not impact the speed of reading data in fully associative caches more circuitry is required to check each line for the data in parallel. A fully associative cache avoids conflict misses and usually results in fewer overall cache misses. However, set associative caches usually have a better cost-benefit tradeoff.

To take full advantage of a cache, the number of cache misses needs to be limited since every cache miss means a read from either a slower cache or main memory. There are three types of cache misses: compulsory, capacity and conflict. Compulsory misses, which are unavoidable, occur the first time that data are read in from main memory. A large cache line size, however, can limit the number of these misses since, when one element on a cache line is read in, the other elements on that line are also read. The extra data that are read on the cache line are prefetched data and, if the processor uses them before the cache line is evicted, then some compulsory misses are avoided. Capacity misses occur when a cache is not large enough to hold all the data the processor is currently using to perform calculations. The two ways to avoid capacity misses are to increase the size of the cache or to change a program so the amount of data it uses at a given time fits within the cache. The final type of cache miss is a conflict miss which occurs when there is space in the cache for a line, but it was evicted because the set in which it is stored was full. To avoid these misses, a cache either needs to be made with more lines per set, which is expensive in terms of the transistors necessary, or a program needs to access memory sequentially, which distributes the cache lines being used by a program more uniformly across cache sets.

2.1.1.3 Translation Lookaside Buffer

Another memory structure used to speed the operation of the memory hierarchy is the translation lookaside buffer (TLB), which stores virtual to physical page translations. A virtual address is the address a program uses to access data. The computer, when retrieving the data, translates it into the physical address where the data reside. Virtual addresses have both advantages and disadvantages but are necessitated by the need to share the limited amount of physical memory among multiple processes and provide data protection to each process' data.

The advantages of virtual memory are that code is easier to write, multiple processes can access the same virtual addresses, and a program can address more memory than is physically present. Virtual addresses enable a program to reside in any part of main memory during execution because the physical address corresponding to a virtual address can be anywhere. Therefore, virtual addresses allow a programmer not to have to worry about memory management when writing code. Virtual addresses also allow multiple programs to access the same virtual address because, for each program, there is a separate mapping of virtual to physical addresses. Therefore, a programmer does not need to worry about another program accessing said program's data because virtual memory provides data protection. In addition, virtual addresses allow a program to address more memory than is physically present by allowing access of data on disk as if it were within memory. In this thesis, the concern is only with data sets that fit within memory. However, the cost of virtual to physical address translations is significant, in both space and time and important to the performance of routines that fit within memory [49].

For the proper physical page to be accessed when a virtual address is referenced, a translation between virtual and physical pages must occur. For each process, therefore, there must be a virtual to physical page table to translate addresses for proper access. The page table is stored within memory, and, when the tables for all running programs are combined, they can take up a significant portion of memory. A virtual address must be translated to a physical address by looking up the physical address in the page table. Each of these lookups requires at least one additional

memory access. For multilevel page tables, which are found on most machines, two or more reads are required per translation. The TLB, by storing recent translations, reduces the need to go to memory to translate addresses and, as with caches, reduces the amount of memory traffic.

A TLB typically contains 32 to 1024 entries, where each stores the translation for a single page of memory. Pages typically contain 4KB to 2MB of data [52]. Most TLBs are fully associative as the added cost of a fully associative structure is offset by its ability to reduce the number of page table lookups. When a miss occurs in the TLB, the cost is amortized if there is good spatial locality among reads since each page contains more data than a cache line.

2.1.1.4 Other Memory Sub-system Structures

In addition to efficient use of caches and the TLB, hardware prefetching improves performance. To reduce the latency of reading data from memory, the next data element to be used is predicted. Then, before that datum is accessed by the program, the hardware begins reading it from memory. If the prediction is correct, prefetching reduces the time that it takes for the datum to be read from memory. The memory prefetcher has many choices for predicting the next accessed data element but typically fetches data consecutive to a current cache miss.

Another hardware feature that improves the performance of the memory subsystem is non-blocking caches. A non-blocking cache allows reads of data within cache to be serviced while a previous cache miss is serviced from memory. By not stalling on a cache miss, the latency of reading data from memory is hidden as other data are read from cache while the memory read is being serviced. Hardware support for out of order execution, which is explained in section 2.1.3, is needed to take advantage of a non-blocking cache.

2.1.1.5 Putting it all Together

The total cost of moving data through through the memory hierarchy can be expressed neatly as shown by Byna et al. [20]. For a memory hierarchy with k levels of cache (called L1, L2; ..., Lk) and one TLB, the amount of time spent moving data can be calculated using the following

equation [52]:

$$\begin{aligned} TotalMemoryCost =& (NumberofTLBhits * TimetoaccessTLB) \\ &+ (NumberofTLBmisses * TLBmissPenalty) \\ &+ (NumberofL1hits * TimetoaccessL1) \\ &+ (NumberofL1misses * L1penalty) \\ &+ (NumberofL2hits * TimetoaccessL2) \\ &+ ... + (NumberofLkmisses * Lkpenalty) \\ &+ (Numberofmemoryhits * Timetoaccessmemory) \end{aligned} \tag{2.1}$$

Byna, et al. also show that, if memory behavior of a sequence of operations can be predicted and the performance of the sequence of operations is limited by data movement, program runtime can be approximated without running the program. In this thesis, we build upon this work and create a memory model for more fused linear algebra routines that combine multiple operations into one calculation.

2.1.2 Multi-core Architectures

Current microarchitecture design has moved from increasing the number of instructions that can be executed by a single core in a given amount of time towards keeping the speed of the cores on a processor fairly constant while increasing the number of cores. Multi-core processors are single circuits that contain two or more processing cores, each with the full computing capabilities of a normal processor. There are two reasons leading to this shift in design. The more important is heat dissipation. Approximately every two years, the number of transistors that fits on a piece of silicon doubles, but the amount of power each transistor uses decreases at a slower rate [67]. Therefore, power consumption per chip is increasing, resulting in more heat on a processor. The other reason

for a shift in design is the problem with synchronizing operations across the processor because, at current clock speeds, it takes an electronic pulse more than one clock cycle to move across a processor. Multi-core processors are the most common solution to the heat and synchronization problem since they decrease power consumption, which reduces heat production, by performing the same number of operations at a lower clock speed. The lower frequency and decreased size of the multi-core processors also make it easier to synchronize chips.

Most multi-core chips are combined together into shared memory parallel (SMP) systems that have multiple sockets each with a bus to memory. The processors on these sockets typically have multiple cores that all share the same memory bus. For example, the dual socket quad-core Intel Clovertown shown in Figure 2.1 has two buses from memory, one per socket. The added buses per socket increase overall memory to processor bandwidth available to perform calculations. However, sharing buses between cores can decrease available memory bandwidth per core and create contention for the memory bus.

In contrast to memory buses, which typically only increase with the number of sockets, the number of caches and buses from cache to the processor often increase more rapidly. In our Clovertown example, there are four L2 caches and eight L1 caches. Additionally, each core has a bus from both an L1 and L2 cache dedicated to it. Therefore, when compared to using a single core of the Clovertown system, memory to processor bandwidth is increased by a factor of two when using the whole system, while cache to processor bandwidth is increased by a factor of eight.

Another positive of parallel systems is that additional caches increase the total amount of data that can be stored close to the processor. However, when caches are shared among cores, as the L2 cache is in our Clovertown system, the amount of each cache a given core can use is reduced. Contention for the cache can cause data to be evicted, resulting in more costly reads from slower memory structures.

Additionally, the organization of the memory sub-system of some SMP computers varies. For some machines all the data stored in memory can be accessed with the same latency and bandwidth by all processors. In the other common memory organization access reads of the same

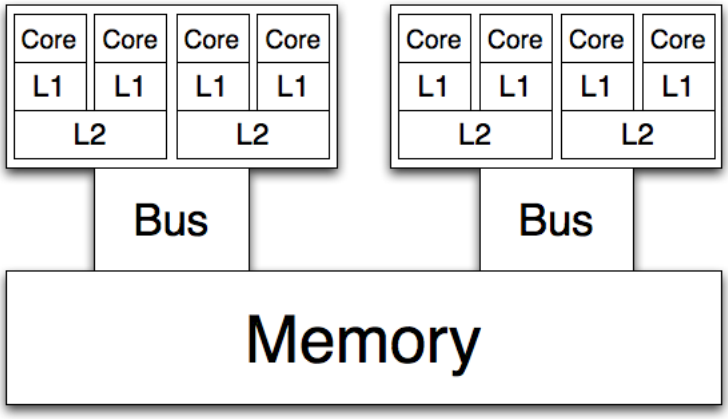


Figure 2.1: Dual socket quad-core Intel Clovertown

data by different processors result in different latency and data throughput rates. In these Non-Uniform Memory Access (NUMA) systems each processor has its own memory address space, and reads from a processor's own address space are quicker than those from other processors' address spaces. Therefore, keeping data reads local to a processor's own address space can increase routine performance.

2.1.3 Processor Memory Speed Gap

The speed of processors is increasing at a greater rate than the speed data are moved from memory into the processor [52, 81]. The result of this differing pace of performance improvement is that, for each word read in from memory, the processor can perform multiple calculations. There are many ways in which computer designers attempt to increase the amount of data moved to the processor per calculation. Caches allow algorithms with good temporal locality to reuse data read from memory more than once, thereby improving the ratio of calculations to memory reads.

To allow the processor to still perform work while waiting for high latency cache or memory reads, out-of-order execution allows a processor to execute later instructions while the current instruction is stalled. The processor analyzes the instruction(s), to be executed after the stalled instruction and any instruction not dependent on the result of any stalled instruction is executed while the cache or memory read is serviced. Through out-of-order execution, the latency of a memory read is mitigated or hidden as the computer performs useful work while waiting for a data read to be serviced. Out-of-order execution, therefore, can increase the total throughput of a computer by eliminating the need for a processor to stall on high latency reads. However, out-of-order execution is limited by the number of instructions that can be stalled at once and how many instructions in the future the processor can examine for calculations that can be executed immediately.

Caches and out of order execution are effective at increasing the overall throughput of programs with good spatial and temporal locality and few latency bound reads. However, they are not effective at increasing the speed of algorithms that do not reuse a datum once it is read from

cache and that have many reads from high-latency or low-bandwidth memory. Such algorithms are inherently limited by the bandwidth from memory to the processor [3]. For algorithms with little data reuse, the only way to increase their speed is by reducing memory operations [30] or combining two algorithms that access the same data [55].

2.2 Performance Tuning Techniques

Naively designed linear algebra routines take longer to execute than highly optimized routines although both should produce the same result to within machine precision. The performance gap occurs because optimized routines take into account the underlying hardware on which the program runs. An example of two routines that perform the same calculations at dramatically different speeds are the matrix-matrix multiply routines of Netlib BLAS [78] and GotoBLAS [22]. The BLAS provide an application programming interface (API) developed to allow application developers a standardized interface to call highly optimized linear algebra routines. Each is an implementation of the BLAS [36,37,71] routine _GEMM. The Netlib routine is an untuned reference BLAS implementation written in Fortran. GotoBLAS is a highly tuned BLAS implementation written primarily in assembly. The top left graph in Figure 2.2, shows that GotoBLAS matrix-matrix multiplication outperforms Netlib BLAS matrix-matrix multiplication by 25% to 900% more MFlops across a wide range of matrix sizes even though each performs the same number of floating-point operations. The remaining three graphs in the figure show the ratio of flops performed to L1, L2, and TLB misses. Executing GotoBLAS results in significantly fewer of these costly misses for large matrix orders, which dramatically improves their performance.

In this section, we discuss the tuning techniques that are used by GotoBLAS and other codes to achieve performance beyond that of a naive implementation. Techniques are grouped into four categories to reflect the portion of the hardware where they improve program performance: faster instructions, pipeline, register and memory. When a tuning technique positively affects more than one hardware structure we include them in its description. For such techniques, the technique is grouped where the largest effects occur. Techniques that are of primary interest in this thesis are

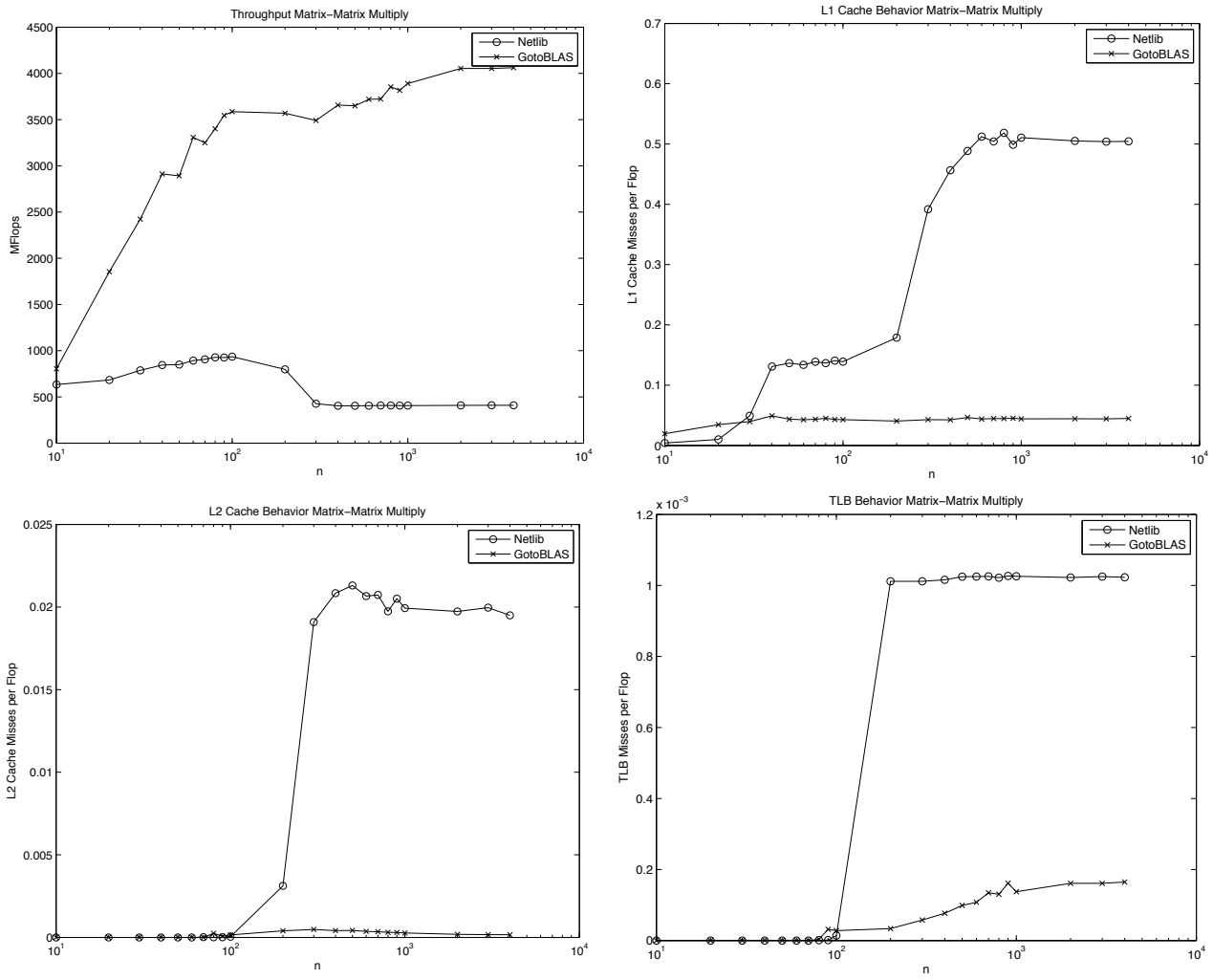


Figure 2.2: Optimized vs. un-optimized matrix matrix multiply performance.

given a full explanation rather than a quick mention and citation. Loop fusion, which is a memory optimization and the primary focus of this thesis is left to the next chapter.

2.2.1 Use Faster but Equivalent Instructions

Some operations can be performed by a computer in multiple ways to yield the same result. For example, $5*2 = 10$ can be computed by shifting 5 one bit to the left, multiplying $5*2$ or adding $5 + 5$. Although each instruction yields identical results, bit shifting is preferable as it executes in the least amount of time by most CPUs. Also, converting integer multiplies to adds when possible results in equivalent code that executes in less time. Eliminating magnitude compares and replacing them with inequality or equality compares results in faster execution. For best performance, pointer updates should be minimized to allow the use of register plus offset addressing mode [13].

2.2.2 Pipeline

Flushing the pipeline because branches are mispredicted or instructions need to be delayed while waiting for the results of other instructions slows the execution of programs. Mispredicted branches can be reduced by minimizing the number of branch instructions. By exposing independent operations through the use of local variables and false dependency elimination, more instructions can be executed by the processor's out of order execution unit at a given time, thereby keeping the pipeline busy [13]. In addition, by unrolling loops [104] and balancing the instruction mix [13], the computer has more choices when scheduling instructions, decreasing the likelihood of the pipeline's idling.

2.2.3 Register

Once data are stored in registers, they are within the fastest memory structures in the computer. Therefore, using a value as much as possible while it is in a register improves performance. Two ways to keep a value within a register are register blocking and the use of local variables. Register blocking alters loops so that each iteration of a loop uses all the registers and therefore

maximizes the reuse of them. Using local variables in code eliminates false read-after-write dependencies. False read-after-write dependencies occur when a compiler (due to indirect accesses) cannot discern if a data element being written to memory will be used by a future instruction. When the compiler wrongly predicts a data element is going to be read again, that data element must be written out to memory when it could remain in a register [13].

2.2.4 Memory

To reduce memory reads and the effects of high latency and low bandwidth, programs can be rearranged. Blocking is used in matrix-matrix multiplications to reduce memory traffic by performing as many calculations as possible on data read from memory before the data are evicted from cache. Figure 2.3 shows how matrices are blocked when they are multiplied. The block sizes are chosen such that one block of A and one block of B fit within cache at once and C is streamed through memory. The operation is performed by multiplying each block in a row of A by each block in a column of B yielding a partial block of C. The process continues for all block rows and block columns until C is produced. The result is a reduction in the number of reads from memory from $O(n^3)$ to $O(n^3/blocksize^2)$. However, careful choice of blocksize is important to performance as shown by Lam et al. [70]. Blocking can be applied to all levels of the cache and the TLB and is usually done for each memory structure [49].

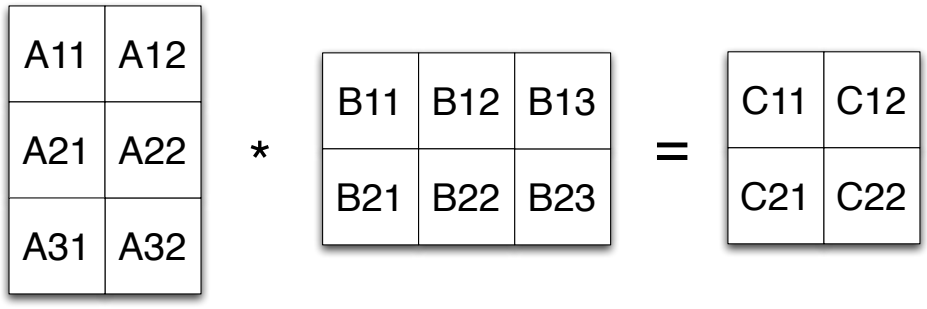


Figure 2.3: Blocking a Matrix-Matrix Multiply.

When blocking is applied to matrix-matrix multiplication routines written in Fortran, two

matrices are accessed by reading down their columns and one matrix is accessed by reading across its rows. In Fortran, where data are stored in a column-wise manner, reading elements row-wise results in bad spatial locality of reads. Through the use of a copy optimization, data elements in the matrix can be rearranged so they are accessed in a consecutive manner [70]. Copy optimizations can be used on blocked matrices to rearrange data within blocks so that they are stored consecutively and that successively accessed blocks follow each other. The combination of blocking and copy optimizations reduces the number of memory reads and improves memory access patterns.

To reduce the number of conflict misses caused by data accessing the same set within cache, array padding [87] can be used. Array padding is a technique in which more memory is allocated to an array than is necessary [38]. Extra data are added to the end of each row or column of a matrix thus separating the last value of the previous row or column from the first of the next with empty space. The result is that data values map to different sets in cache, reducing conflict misses.

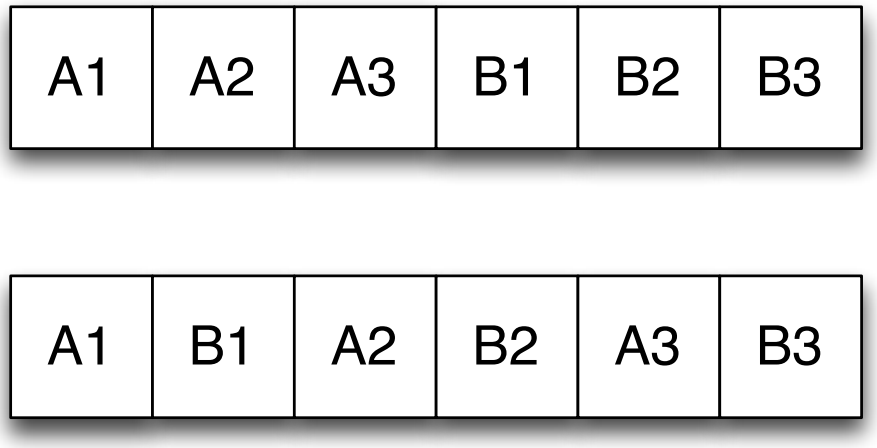


Figure 2.4: How to interleave data to create a multivector.

Data interleaving, also called multivector optimization, reorganizes multiple vectors to improve the data access pattern of the vectors [66]. The goal is to decrease the number of cache lines accessed and thus the potential for conflicts. Figure 2.4 shows how two arrays can be interleaved. The top image in the figure is the data in memory where the two vectors are declared separately.

The bottom image is the data in memory where the elements of each vector are interleaved. To perform a multivector optimization the first element of each vector, A_1 and B_1 in our example, are moved to successive memory addresses. Then the second element of each vector immediately follows the first in the same order. The process is repeated with the locality rest of the elements in each vector resulting in a single multi-vector composed of independent vectors. If both vectors a and b are accessed consecutively within a loop then all the data are accessed consecutively improving spatial locality.

Another way to reduce the number of reads from memory is software pipelining. Software pipelining interleaves operations where a dependency exists. It reorders dependent operations such that they are overlapped in code [2]. An example of software pipelining is shown in figure 2.5. The left calculation is the unpipelined routine and the right calculation is the pipelined one. The software pipelined routine reduces the number of times the vectors b and c need to be read from memory by one since all accesses occur within a few calculations of each other rather than a loop apart.

```
for  i = 1 : n                      b(1) = b(0) + c(0)
   b(i) = b(i − 1) + c(i − 1)       c(1) = c(0) + c(−1)
   c(i) = c(i − 1) + c(i − 2)       for  i = 2 : n
for  i = 1 : n                         b(i) = b(i − 1) + c(i − 1)
   a(i) = b(i + 1) + c(i + 1)          c(i) = c(i − 1) + c(i − 2)
                                       a(i − 1) = b(i) + c(i)
                                    a(n) = b(n + 1) + c(n + 1)
```

Figure 2.5: Software pipelining of a calculation

2.3 Linear Algebra Libraries

Since many scientific computing applications and other computing problems use similar sets of linear algebra calculations, libraries are designed to allow the reuse of codes that perform those calculations [4, 7, 15]. Routines are written for various linear algebra structures and environments including dense matrices, sparse matrices and parallel processing environments. Many approaches

use the same API to allow for an application developer to design one program and link in the proper optimized library for the machine on which the program is run. In addition, many packages build on each other to allow for code reuse and modularity. The result can be highly portable code that performs efficiently across many machines for many commonly used operations. In this section, we discuss major libraries and APIs that perform linear algebra computations and discuss their importance in scientific computing.

2.3.1 Basic Linear Algebra Subprograms

The BLAS are a standard Fortran interface for dense linear algebra operations [36,37,71] that has been updated to include sparse operations, additional routines and a C interface [15, 34, 35]. The BLAS are broken into three levels of operations. Level 1 routines contain vector operations, such as an inner product, that perform $O(n)$ computation on $O(n)$ data. Level 2 routines are matrix-vector operations, such as outer products and matrix-vector multiplies, that have $O(n^2)$ computation on $O(n^2)$ data. Level 3 routines carry out matrix-matrix operations, such as matrix-matrix multiplications, where $O(n^3)$ computation is performed on $O(n^2)$ data. With the updated standard, four new routines that perform the work of multiple old routines and a sparse BLAS (SBLAS) interface which provides a standard interface for a variety of sparse matrix storage formats [39] are added.

The BLAS were first developed to provide linear algebra operations to application developers through a standardized interface. Before their creation, application developers would create their own code to perform linear algebra routines for each program. Since the development of tuned BLAS implementations application developers are able to create a portable program that, when linked with a tuned BLAS library on the target machine, delivers high performance. Another benifit of the BLAS is that often tuning linear algebra routines must only be done once for each architecture, by library developers, rather than by each programmer.

There are three types of BLAS implementations available: reference, handcoded and machine generated. The Netlib BLAS [78] are a reference dense BLAS that other developers can use to

compare their implementations against for correctness. The library contains slightly optimized or unoptimized versions of the routines defined in the BLAS specifications [15, 36, 37, 71]. Reference sparse BLAS are available in Fortran from Netlib [79] and C++ from NIST [80]. The Netlib BLAS also provide a C interface to ease the development of programs written in C and C++.

Handcoded BLAS routines are highly optimized routines that typically use assembly code to perform the most important parts of their calculations and as a result are among the fastest BLAS implementations available. Most computer vendors supply their own versions for their machines. Some examples are Engineering Scientific Subroutine Library (ESSL) [56] for IBM processors, Math Kernel Library (MKL) [57] for Intel processors, and Sun Performance Library [96] for Sun processors. Some of these libraries, such as MKL, also include optimized sparse computations. In addition to vendor supplied packages, Kazaski Goto formally of the Texas Advanced Computing Center developed and distributed GotoBLAS [22], another highly tuned BLAS package available for multiple architectures.

The problem with handcoded BLAS routines is that they are expensive to create. An alternative is the automatic generation of kernels. Two programs, Automatically Tuned Linear Algebra Software (ATLAS) [105] and Portable High Performance ANSI C (PHiPAC) [13], generate high performance dense linear algebra algorithms for many architectures. In addition, OSKI [98] generates high performance sparse kernels and can perform runtime tuning by transforming data structures. All three programs search through a range of parameters to find the best settings for a given machine. They can either figure out cache sizes or be given machine parameters to shorten the amount of time spent searching for them. The result is performance that is comparable to or better than handcoded operations, but with less programmer effort. While very good at producing highly efficient multi-platform code, these packages need constant updating to take advantage of new instructions and architectures as they appear.

Since most scientific simulations are run on parallel machines, parallel BLAS routines have been developed. ATLAS, GotoBLAS, and most vendor BLAS have multi-threaded implementations, and a multi-core aware version of OSKI was considered [99]. The multi-threaded libraries

are used on SMP machines. For distributed memory machines, Parallel BLAS (PBLAS) [24] are used. PBLAS calls the Basic Linear Algebra Communications Subprograms (BLACS) [32] to handle communication between nodes. In addition, the author of this thesis implemented an interface to OSKI within Epetra [90], a linear algebra library, that allows for most of OSKI's kernels to be used for serial computation within parallel routines [64].

Furthermore, implementations of the new routines contained in the updated BLAS standard [15] have resulted in significant speedups [55]. These routines combine the functionality of two or more level 1 or level 2 BLAS routines into a single more memory-efficient routine. They have been used by Howell et. al to improve the speed of Householder bidiagonalization by 10% to 25% [55].

2.3.2 Linear Algebra Package (LAPACK)

Once the BLAS were designed and widely available, it became practical to have higher level packages that use them to perform more complex calculations. The first of these packages were EISPACK [45, 94], which includes routines for computing eigenvalues and eigenvectors of matrices, and LINPACK [33], which is used to solve linear equations and linear least-squares problems. Both EISPACK and LINPACK were written in Fortran and use column operations and functions in the level 1 BLAS. The advantage of having one combined package, the ability to optimize those routines through calls to the more efficient level 3 BLAS routines, and algorithm advances led to the creation of the Linear Algebra Package (LAPACK) [4]. Since it uses level 3 BLAS routines, LAPACK takes better advantage of the memory hierarchy, which for most machines results in more efficient computation than the level 1 BLAS routines used by its predecessors.

A parallel version of LAPACK called Scalable LAPACK (ScaLAPACK) [14] is available. To increase code reuse, ScaLAPACK uses PBLAS and BLACS for computation and communication whenever possible.

2.3.3 Higher Level Packages

Leveraging the BLAS and LAPACK are libraries such as Trilinos [54] and PETSc [7]. These packages build upon the BLAS and LAPACK by providing data structure support and higher level solvers and preconditioners. Trilinos is an object-oriented framework for the solution of large scale complex physics, engineering and scientific problems. The project is designed to assist the development of independent packages which are autonomous pieces of code that can function independently or, using support provided by the Trilinos framework, communicate with each other. Examples of such packages and their functions are Sacado [91], which performs automatic differentiation, ML [46], a multigrid preconditioning package and Amesos [89], a direct sparse linear solver package.

PETSc provides many sparse linear solvers, handles communication for the programmer and provides automatic profiling of floating-point and memory usage, all within a consistent interface. In addition, related projects that use the same data structures and some PETSc routines provide added functionality. TAO [74], the Toolkit for Advanced Optimization, adds in the ability to solve optimization problems. SLEPc [53], the Scalable Library for Eigenvalue Computation, is used for solving eigenvalue problems. Finally, Prometheus [1] is a multigrid solver used to solve unstructured finite element problems.

The importance of the speed of linear algebra computations to overall runtime of higher level libraries and application codes is shown by the constant effort made to improve the runtime of the underlying linear algebra. The OSKI interface within Epetra is now available to all Trilinos package developers and allows them another option when looking for the fastest available routines. In addition, the author of this thesis worked on improving the speed of a matrix-matrix multiply within ML by creating a block version of it [63]. The constant search for faster algorithms and methods within Trilinos demonstrates that improvements in the speed of linear algebra routines are both sought and used in real scientific applications.

2.4 Autotuning Programs

Autotuning is used to reduce programmer effort by having a program self tune based on a set of rules. Efforts for diverse calculations, such as matrix multiplication and Fast Fourier Transformations have shown success in producing routines that are faster than hand-tuned libraries. In this section, we examine other auto-tuned linear algebra software.

ATLAS [105] and PHiPAC [13] both produce matrix-matrix multiplication kernels that are are competitive with hand-tuned codes across various architectures. Both of these systems query the hardware to determine characteristics of the target machine and produce optimized codes during installation. ATLAS produces multi-threaded code that runs on parallel machines.

OSKI [98] takes a different approach. It focuses on sparse kernels and matrix vector operations. It profiles the hardware at install time, but performs runtime tuning because the proper tuning techniques for sparse kernels are not known until the matrix structure can be determined. Therefore, by deferring tuning until runtime, more specialized optimization can be performed for the target matrix. The drawback of tuning at runtime is that the tuned kernel needs to be called many times to amortize the tuning cost [64].

FFTW [42] and SPIRAL [84] both perform discrete Fourier transforms. In addition, SPIRAL performs other transforms used in digital signal processing such as discrete cosine transformations. FLAME [51] partially automates the process of generating provably correct linear algebra algorithms.

2.5 Models

Memory models are used to estimate the numbers of reads and writes that will occur from each level of memory when a program is executed without running the code. Through the use of such models, the memory behavior of different algorithms can be predicted. From these predictions, the runtime can then sometimes be estimated, for example, by using Equation 2.1. In this section, we discuss memory models that have been implemented including what they estimate and what

they do not. The models are divided into two sections, those that predict performance and those that give performance bounds. The models presented are chosen for their diversity and influence on this research.

2.5.1 Memory Models that Predict Performace

Some memory models attempt to approximate how many misses occur in each memory structure and use those estimates to predict runtime performance. Predictions can aid developers in creating code and can lead to large reductions in program runtimes. Here, we summarize models that attempt to project the performance of calculations they model.

Byna et al. [20] predict the amount of time each memory reference in a segment of code takes. They analyze the cost of differeny access patterns including sequential and strided. These costs and their memory predictions are used to estimate the amount of time two matrix transpose algorithms take.

The PAD/PADLITE model is designed to approximate conflict misses and then minimize them through array padding [87]. It uses the Euclidian algorithm for computing the greatest common denominator to calculate conflicts between array columns in linear algebra code. Padlite is the simpler version of the model and is only effective for eliminating conflicts in benchmark programs. Pad is a more complex version that can improve performance for more complicated kernels. The heuristics used lead to large improvements for some problems and sizes and minimal to no improvement for others. The largest gains come for arrays where the size has a large power of two as a factor.

The Sparse Linear Algebra Memory Model (SLAMM) [29,30] uses automated memory analysis to speedup iterative methods. Using compiler techniques, SLAMM is able to analysis a MATLAB implementation of an algorithm and determine the minimum amount of data movement necessary to implement the algorithm in C or Fortran. SLAMM proved effective by showing that the conjugate gradient (CG) solver in the Parallel Ocean Program (POP) had more memory traffic than necessary. Through the use of SLAMM, memory traffic for CG was reduced by 18% resulting in

a 46% speedup in the runtime of CG. The speedup in CG resulted in an overall 9% reduction in the runtime of POP and a savings of 216,000 CPU hours per year at the National Center for Atmospheric Research (NCAR) [30].

Ghosh et al. [47] formulate equations for reuse distances and cache misses that provide a high degree of accuracy. Their model almost exactly predicts cache misses when compared to actual misses provided by hardware counters. While accurate, this approach suffers from a large cost of creating and evaluating the equations.

Yotov et al. [108] develop an analytic model for matrix multiplication. They show that their analytic model can produce codes that perform almost as well as the automatically generated routines from ATLAS. Their model takes approximately half the time to run as ATLAS does and produces routines that perform as well to less than 10% worse than the ATLAS generated kernels.

2.5.2 Models that Generate Performance Bounds

Other memory models do not attempt to generate accurate predictions but rather generate upper and/or lower bounds on performance. Using these bounds, which can be the number of misses to a structure, runtime or MFlops, an estimate of performance is obtained. The following is a summary of models that provide performance bounds in their predictions.

Ferrante [41] presents a capacity miss model that calculates the amount of data accessed by a series of nested loops. It computes a bound on how many cache lines are used by these data. Validation tests performed on matrix-matrix multiplications show that the model can predict when to interchange loops to decrease the amount of data accessed within inner loops.

In [88], Rivera and Tseng present models that leverage their previous work on Pad to predict tile and array pad sizes for 3D stencils. Rivera and Tseng's model is then used in [27] on SMPs and other new microprocessors to predict upper and lower bounds of execution time of algorithms. For the lower bound only compulsory misses are considered while the upper bound is calculated from the maximum amount of data movement that could occur during execution. The model is effective as it bounds the actual runtime in all but one case.

2.6 Hybrid Search

Many authors combine models and other search techniques to optimize routines. For example, Yotov et. al [109] use analytic methods to perform global search and approximate the values that should be used for tuning parameters. They then use empirical search to fine tune their estimates and further improve performance. The combined search results in better performing routines than only using a model in significantly less time than global search.

Chen et al. [23] use analytic modeling and heuristics to select a small number of optimization variants. Combinations of loop permutation, unrolling, register and loop tiling, copy optimization, and prefetching are selected. Empirical testing is used via guided search to select the best variant, typically in three to eight minutes. The resulting performance is comparable to vendor BLAS and ATLAS implementations of matrix multiplication.

Epshteyn et al. [40] consider loop tiling decisions in the context of matrix multiplication. They use an explanation-based learning algorithm to adapt their analytic model based on empirical results. The result is faster code than modeling or empirical search alone in comparable or less time than ATLAS's empirical search and somewhat more time than the modeling used by Yotov.

Qasem [85] uses pattern-based direct search to find good combinations of loop fusion decisions for Fortran programs. A model guides the search direction of his compiler for these decisions. Since Qasem targets a general purpose language and compiler, he needs the scalability of direct search to produce tractable compile times even though direct search sometimes misses the globally optimal fusion decision.

Chapter 3

Loop Fusion

Loop fusion is a memory optimization that combines two loops that access the same data into one [44]. For calculations where memory bandwidth is the limiting factor on performance, fusing loops can reduce data traffic through the memory subsystem and decrease runtime [28, 55, 101]. However, fusing loops can decrease performance if not done carefully. Fusion can increase the amount of data stored in a cache or registers leading to capacity and conflict misses [60, 61]. Also, determining which fusion decisions result in the best performance is non-trivial as the search space of all possible fusion decisions is NP-complete [26].

In this chapter, we first overview other research using loop fusion to improve the speed of linear algebra calculations. Then, we show how to use loop fusion to reduce memory traffic. A detailed overview of the BTO compiler follows, including how we leverage it to improve our memory model and learn more about loop fusion. Next, we discuss how negative memory effects can result when loops are fused. How the memory subsystems interacts with the fused calculations' data access patterns to decrease performance is explained. We then describe how to combine other tuning techniques with fusion to mitigate these negative memory effects. Finally, we present the search complexity of finding the optimal amount of fusion. In our discussion of the complexity we present work that shows ways to limit the testing of fused routines without reducing routine performance.

In this chapter and the rest of the thesis tests were run on various computers. These computers and their associated architectures are listed in Table 3.1. Throughout the thesis they are

Name	Processor	Speed	Mem	Bus Speed	L1	L2	L3	TLB
Hemisphere	Intel Xeon	2.4 Ghz	2 GB	400 MHz	8 KB	512 KB	N/A	64
Quadfather	Intel Core 2	2.4 GHz	4 GB	1066 MHz	32 KB	4 MB	N/A	256
Work	Intel Core 2	2.4 GHz	2 GB	1333 MHz	32 KB	4 MB	N/A	256
Opteron	AMD Opteron	2.6 GHz	3 GB	1000 MHz	64 KB	1 MB	N/A	40/512
PowerPC	PowerPC 970FX	2.3 GHz	8 GB	1150 MHz	32 KB	512 KB	N/A	1024
Nahalem	Intel Core i7	2.8 GHz	4 GB	1333 MHz	32 KB	256 KB	8 MB	64/512

Table 3.1: Specifications of the test machines. For TLB, we list the number of entries.

referred to by the description in the name column of Table 3.1. Also, various mathematical kernels were used to perform tests. Table 3.2 contains the operations performed in these kernels. Throughout the thesis some of these kernels are presented in more detail to illustrate points.

3.1 Loop Fusion Applications to Linear Algebra

To improve the speed of memory bound linear algebra routines, many authors have used loop fusion on a diverse set of calculations [16, 18, 55, 82, 83, 101]. The work has ranged from optimizing a single routine to general purpose tools within compiler frameworks. The result is significant reductions in runtime for the routines. In this section, we present a survey of tools and approaches used to create loop fusion routines. The techniques are broken into those focused on extracting the most performance possible from a single routine or a few routines and general approaches designed to handle a large number of routines well.

3.1.1 Optimizing Specific Routines

Many frequently used sequences of calculations can be combined using loop fusion. For these calculations, such as the four fused routines recently added to the BLAS [15], spending a large amount of time optimizing them is worthwhile. Two of these routines `_GEMVT` and `_GEMVER` are used by Howell et. al [55] to speed up the execution of the Golub and Kahan algorithm to perform Householder Bidiagonalization [48] by up to 25% as compared to the LAPACK implementation `_GEBRD`. Since Householder Bidiagonalization is half the cost of computing a singular value decom-

Kernel	**Operation**
AXPYDOT	$z \leftarrow w - \alpha v$
	$r \leftarrow z^T u$
AATX	$y \leftarrow AA^T x$
BiCGK	$q \leftarrow Ap$
	$s \leftarrow A^T r$
DGEMV	$z \leftarrow \alpha Ax + \beta y$
DGEMVT	$x \leftarrow \beta A^T y + z$
	$w \leftarrow \alpha Ax$
GEMVER	$B \leftarrow A + u_1 v_1^T + u_2 v_2^T$
	$x \leftarrow \beta B^T y + z$
	$w \leftarrow \alpha Bx$
GESUMMV	$y \leftarrow \alpha Ax + \beta Bx$
GRAMM	$r \leftarrow q^T * v$
	$v \leftarrow v - r * q$
HOUSE	$A \leftarrow A - \alpha v * (v^T A)$
VADD	$x \leftarrow w + y + z$
WAXPBY	$w \leftarrow \alpha x + \beta y$

Table 3.2: Kernel specifications.

position (SVD) [48] in LAPACK and SVD is a frequently used algorithm, fast implementations of _GEMVT and _GEMVER are important enough to warrant the time investment into fast performing implementations. Also, Premkumar shows how to efficiently parallelize _GEMVT [83].

Vuduc et. al present an efficient sparse implementation of $b = A^T Ax$ where they combine loop fusion and other performance tuning techniques to produce fast performing routines [101]. This calculation appears in linear programming and linear least squares [68, 103]. The same fusion strategy can be used to compute $b = AA^T x$, which occurs when using Kleinberg's algorithm to find authorities in hyperlinked environments [69]. These two fused kernels, $b = A^k x$ and the fusion of two matrix vector multiplies, are implemented in the Optimized Sparse Kernel Interface (OSKI) [98, 102]. OSKI is an auto-tuning package that makes tuning decisions at runtime based on user input parameters. The calculation $b = A^k x$ appears in s-step methods [25] while the two matrix vector multiples occur in the bi-conjugate gradient method [8]. OSKI increases the speed of parallel sparse block matrix-vector multiplies within the EPETRA package of Trilinos [64]. It is also included within the Portable Extensible Toolkit for Scientific Computation (PETSc) [106].

3.1.2 General Tools for Fusion

Fusing and tuning one or a few routines at a time is costly. Constantly changing computing architectures makes the challenge greater. Updating routines for new platforms is costly and only worthwhile for frequently used routines. Specialized compilers reduce the time and effort to fuse other routines.

PLUTO, which performs automatic parallelization and locality optimization for multicores using the polyhedral model, also performs loop fusion [16, 18]. PLUTO produces speedups of up to 100% over vendor-tuned BLAS implementations and vendor compilers. Runtime options allow a user to have PLUTO fuse as many loops as possible or use heuristics to determine the optimal amount of fusion to apply. Unfortunately, the heuristics are not published so there is no way to measure their usefulness other than a posteri. Additionally, results from their website show that that their heuristics do not always produce optimal routines [17].

An updated approach using other polyhedral tools along with PLUTO adds iterative search [82]. The result is speedups as compared to using the Intel C Compiler (icc) [58] with auto parallelization on multiple platforms. This approach is limited by tiling only being performed for the L1 cache regardless of profitability. Also by using directed and guided search, local and not global maximums could be found [72].

A similar approach taken by Qasem works on Fortran codes [85]. He uses a model that accounts for all levels of cache to guide empirical search [86]. Qasem also targets a general purpose compiler and uses direct search, which is more scalable than exhaustive search. However, like PLUTO his search can end up in local extrema missing the globally optimal solution.

In Chapter 3.3, we describe our approach of using a domain specific language and compiler to solve the loop fusion problem. Starting in Chapter 4, the remainder of this thesis focuses on how we model loop fusion to reduce the search space of our compiler. Our approach differs from the previous compilers in the following ways: we enumerate all possible loop fusion combinations and then let our model determine a set of the best routines. These routines are then empirically tested to determine the highest performing. Using this approach we are able to find the globally optimal routine. Enumerating all possible loop fusion combinations differs from both the Qasem et. al and PLUTO approaches of using models or heuristics to guide and narrow the search space. By searching through the whole space, we cannot get stuck in local extrema. Additionally, our model and code generation include parallel capabilities missing from Qasem's approach.

3.2 Fusion Reduces Memory Traffic

When properly applied, loop fusion reduces the amount of data that must move through the memory hierarchy to perform a calculation. For memory bound routines, loop fusion can increase performance nearly proportionately to the reduction in memory traffic. In this section, we demonstrate how to fuse independent and dependent calculations and show how each improves the performance of a routine by reducing memory traffic.

3.2.1 Independent Calculations

Independent calculations occur when the result of one calculation does not affect the result of the other. When independent calculations occur in close proximity to each other in an algorithm, they can be combined using loop fusion. Fusing independent calculations is the simplest case because data dependencies do not need to be observed. The two matrix-vector multiplies $Ax = r$ and $A^T y = s$. that occur in the bi-conjugate gradient method [8] are one example.

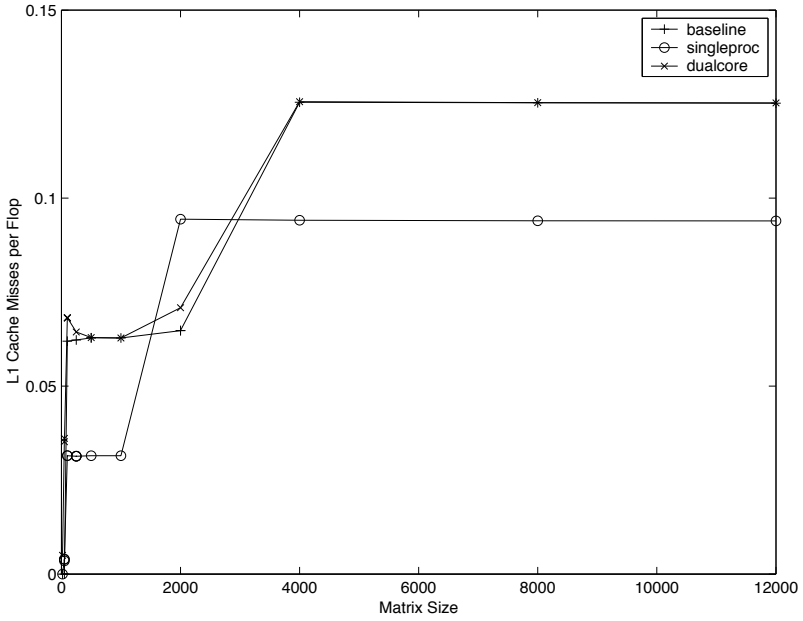
In Figure 3.1 we show how to fuse the calculation $Ax = r$ and $A^T y = s$. By accessing A twice in succession, the value can remain within a register during the calculation instead of being read from a cache or memory. The resulting one half reduction memory reads on the Quadfather machine for large matrix orders is shown by the decrease in L2 cache misses in Figure 3.2. Figure 3.2 also shows that fusion reduces the number of executed loads, and the number of misses to the L1 cache and the TLB. L1 cache misses are reduced by one half for small matrices and one quarter for larger ones. All values are normalized to be per flop. TLB misses which are not shown are reduced by one half for large matrix orders. Additionally, for all matrix orders, there is a reduction by upto one quarter in the number of load instructions executed. The corresponding performance increase to the reduction in data movement of approximately 90% is shown in Figure 3.3.

```
for i = 1:n                          for i = 1:n
  for j = 1:n                          for j = 1:n
    r(j) = A(j,i) * x(i)                 r(j) = A(j,i) * x(i)
for i = 1:n                              s(i) = A(j,i) * y(j)
  for j = 1:n
    s(i) = A(j,i) * y(j)
```

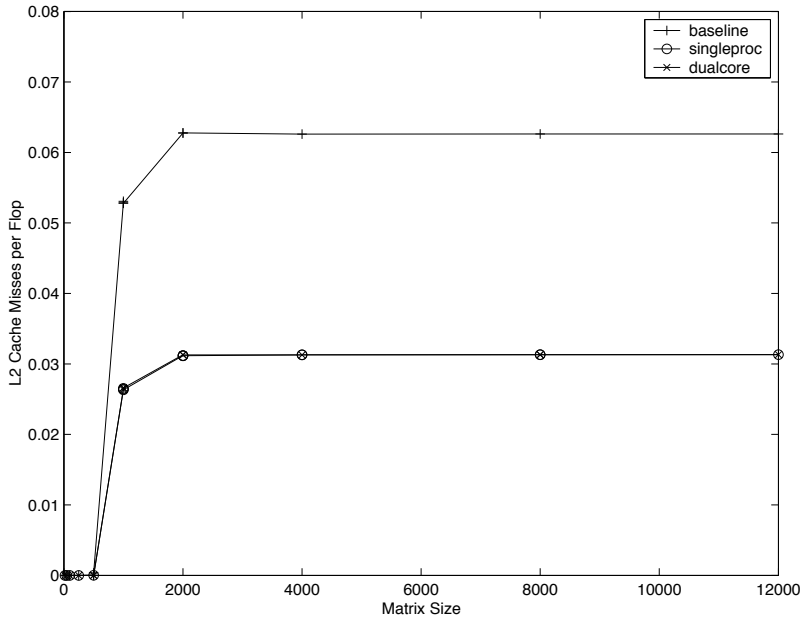
Figure 3.1: Fusing $Ax = r, A^T y = s$

3.2.2 Dependent Calculations

Dependent calculations occur when part of one calculation must be complete before the other calculation can be started. An example of one such calculation is $b = AA^T x$, which mentioned previously occurs when using Kleinberg's algorithm to find authorities in hyperlinked environments



(a) L1



(b) L2

Figure 3.2: Memory performance of fused and unfused $Ax = r, A^T y = s$

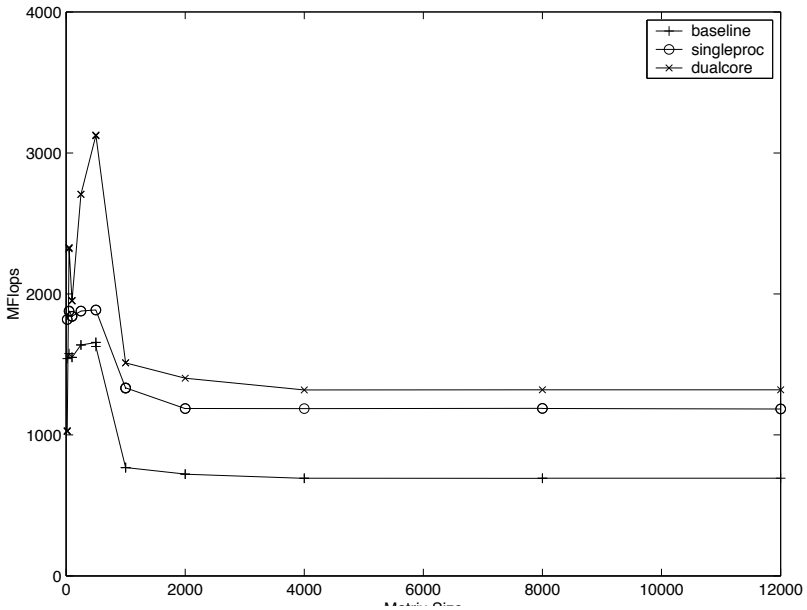


Figure 3.3: Performance of fused and unfused $Ax = r, A^T y = s$

[69]. It is important to note that the same technique used to combine $b = AA^Tx$ is used to fuse the computation $b = A^TAx$ [101].

Fusing loops of $b = AA^Tx$ requires accounting for the dependence between the two matrix-vector multiplies. The matrix-vector product A^Tx is computed as inner products of the rows of A^T with the vector x. Each inner product results in one element of the vector t. The matrix-vector product $b = At$ is then computed via a linear combination of the columns of A with the elements of t as coefficients. Thus, the second matrix-vector product cannot begin until at least one element of the vector t has been computed. As each new element of t is computed, one more column of A can be added to the linear combination. Because the inner product with that column is computed independently of the linear combination, the column is retrieved from cache once for each matrix-vector product.

An important consideration with dependent calculations is that they sometimes require data structures to be aligned for good performance. For example, to efficiently compute a fused version of $b = AA^Tx$ a column major data structure is needed so that the memory accesses for each inner product and linear combination are sequential rather than strided. A corollary is that to perform A^TAx efficiently a row major data structure must be used.

In Figure 3.4 we show how to fuse the calculation $AA^Tx = b$. In this case, only the outer loops of the calculation are fused because of the dependence between the calculations performed in the inner loops. The resulting performance improvement of approximately 60% on the quadfather machine for matrices larger than 1000 is shown in Figure 3.5. The improvement in memory and cache reads are shown in Figure 3.6. Memory reads for the fused calculation are half those of the unfused version. Reads from the L2 cache are the same for both fused and unfused matrices of 3000 and larger. The increased L2 reads are the added cost in performing the fused calculation when compared to the fused dependent calculation. The reason for the added cost is that these reads occur while the memory bus is idle, as explained in the next section.

```
for i = 1:n                    for i = 1:n
  for j = 1:n                    t = 0.0
    t(i) += A(j,i) * x(j)        for j = 1:n
for i = 1:n                        t += A(j,i) * x(j)
  for j = 1:n                    for j = 1:n
    b(j) += A(j,i) * t(i)          b(j) += A(j,i) * t
```

Figure 3.4: Fusing $b = AA^Tx$

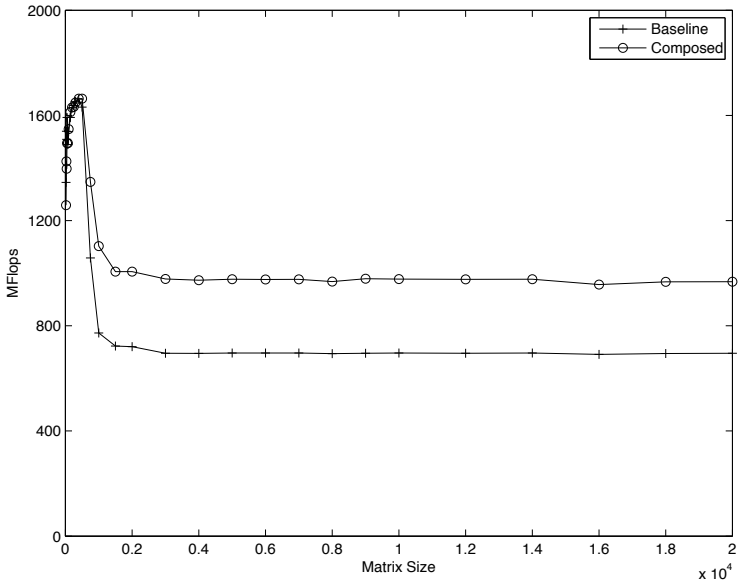
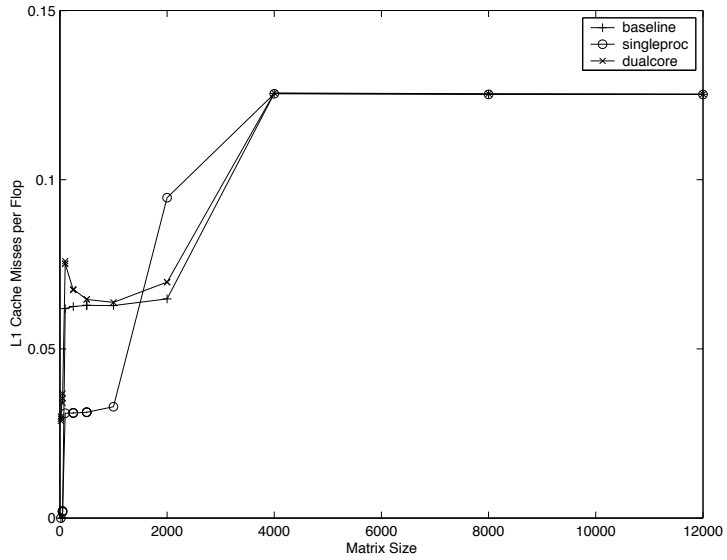
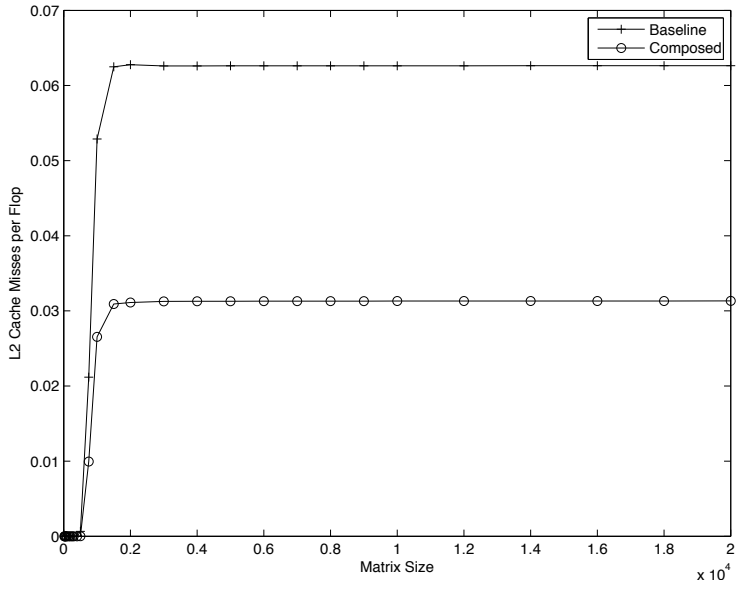


Figure 3.5: Performance of fused and unfused $AA^Tx = b$



(a) L1



(b) L2

Figure 3.6: Memory performance of fused and unfused $AA^Tx = b$

3.3 Build to Order (BTO) Compiler

The BTO compiler [10, 92] is a system that takes in a subset of annotated MATLAB and produces optimized kernels in C. Its primary goal is to create memory-efficient linear algebra kernels for shared memory computers by reducing data traffic through the memory hierarchy. To limit memory traffic, the compiler uses two forms of loop fusion. Data partitioning enables two additional features: cache blocking, which can further reduce data movement, and shared memory parallel codes [11]. BTO ensures the creation of efficient routines by exploring the entire search space of potentially profitable parallelization and optimization decisions. A secondary goal of the project is ease of use. Ease of use is accomplished by automating the creation of efficient linear algebra routines from an accessible high level input.

In this section, we first describe the BTO compiler at a high level. Components relevant to the analytic memory model presented in chapters 4, 5 and 6 are discussed more thoroughly than other components. We also discuss how we use BTO to aid in the testing and development of the memory model. and increase our knowledge of loop fusion.

3.3.1 Functionality of BTO

The BTO compiler works in phases. In the first phase, it parses the input MATLAB and generates a data flow graph of that input. Next, it performs the refinement phase where it generates loops and scalars from the high level input of matrix and vector operations by performing graph lowering operations. The first step during the loop creation and lowering is a data partitioning algorithm that labels some loops to be used to create shared memory parallel code or cache blocks during code generation phase of the compiler. Data partitioning is applied to a single operation in a calculation and then propagated to other data structures that share a dependency with it. Once data partitioning decisions are complete, the compiler then performs graph lowering to generate loops and scalars to carry out the calculations expressed by the MATLAB input.

Next, the optimization phase applies loop fusion to the input routine. First, it enumerates

all potentially profitable combinations of two forms of loop fusion, interleaving and pipelining, that can be applied to the input routine. A fusion opportunity is potentially profitable when the loops share at least one data structure. Interleaving involves fusing loops of two independent operations. In this case, any data accessed by both operations are read once after the routines are fused. Pipelining fuses two operations where one operation consumes the result of another. Pipelining reduces the number of data traversals and removes the need for an intermediate array. Examples of these operations are found in Section 3.2 where independent fusion represents interleaving and dependent fusion is an example of pipelining. Routines fused using pipelining, could benefit from software pipelining as described in Section 3.5.2, however this optimization is not currently included in BTO.

The optimization phase produces multiple versions of the input routine. Each version differs from all others in at least one way. The aspects that vary between routines are the amount of fusion, parallelization, number of cache blocks and size of the blocks. These versions are then passed to the evaluation phase.

In the evaluation phase, all versions of a routine are tested using a two step process. First the analytic memory model is run on all versions of a routine, producing a sorted list of predicted runtimes. Then the best routines are empirically tested with the fastest generated into C code. The interaction between the two steps in the evaluation phase is user-controllable through runtime options explained in Chapter 7, which also presents how the options influence compiler runtime and the performance of the produced routine. After the evaluation phase, the code generation phase outputs the best performing version from empirical testing as C code. For shared memory parallel codes PThreads [76] is used to create separate processes that can be run on different processors in parallel.

3.3.2 Using BTO to Test and Improve the Memory Model

To extensively test the memory model, we leverage the BTO compiler's ability to empirically test all versions of a routine it produces. Using BTO, we can compare memory predictions to

actual memory traffic measured by hardware performance counters across multiple versions of a wide range of routines. Runtime predictions produced by the model are easily compared to actual runtimes of the routines. The large number of versions BTO produces helps ensure that the model can handle diverse input and assists in discovering bugs in the model. When earlier versions of the model were inaccurate, the feedback from these tests showed us ways to improve it. We can evaluate the model's memory and runtime prediction accuracy across a wide number of routines as presented in the next four chapters of this thesis.

3.3.3 Leveraging BTO to Learn More About Loop Fusion

The BTO compiler's ability to enumerate all potentially profitable loop fusion optimizations allows us to run experiments that increase our understanding of loop fusion. We then add this new knowledge to the model to increase its accuracy. For example, we used the compiler's enumeration capabilities to fuse an arbitrary number of matrix-vector multiplies together in a controlled experiment, which we present in Section 3.4. This experiment demonstrates that fusing too many loops together negatively impacts performance. The results of the experiment led us to add registers to the memory model [60], as explained in detail in Section 4.3.2.

Another use of BTO's ability to enumerate all fusion possibilities is that we can test how the fusion of certain operations impacts performance. We use BTO to show that fusing vector operations with matrix operations when the vector is accessed only once never significantly improves routine performance [62]. The results of these experiments shown in Section 3.6 will allow us to decrease the size of the search space enumerated by the compiler, reducing compile times.

3.4 Negative Memory Effects of Fusion

Loop fusion does not always result in increased performance. Decreases in performance occur when fusion creates code that requires more data to fit in a memory structure of the computer than the structure can hold [60,61]. In other cases, loop fusion results in memory access patterns that are sub-optimal [61], for example, non-consecutive reads. In this section, we explore how loop fusion

can negatively affect cache, registers and memory access patterns.

3.4.1 Performance Degradation

In order to observe effects that occur when many loops are fused, we look at what happens when many matrix-vector multiplies are combined. For example, we define a routine DGEMV2 that multiplies vectors u_0 and u_1 in turn by a matrix A as shown in Figure 3.7. The annotated MATLAB provided in this figure serves as input to the BTO compiler which generates all possible loop fusion combinations for the pair of matrix-vector multiplies. Figure 3.8 shows three of these possibilities ranging from the least complex to the most complex: no loop fusion, only outer loops fused, and all loops fused.

```
DGEMV2
in
  u0 : vector, u1 : vector,
  A : row matrix
out
  v0 : vector, v1 : vector
{
  v0 = A * u0
  v1 = A * u1
}
```

Figure 3.7: DGEMV2

3.4.1.1 Cache Effects

If we fuse the outer loops of an arbitrary number of matrix vector multiples on the Opteron system, performance degrades for large sized matrices as shown in Figure 3.9. The dropoffs occur due to increased L2 cache misses as shown in Figure 3.10. For large calculations, the u_x vectors are accessed with each iteration of the outer loop. However, for large matrix orders the combined size of the vectors is larger than cache meaning that they must be read from memory. Increasing the number of vectors accessed per iteration of the outer loop results in a greater number of misses for smaller matrix orders.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    v0[i] += A[i][j] * u0[j]
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    v1[i] += A[i][j] * u1[j]
```

(a) No Fusion

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    v0[i] += A[i][j] * u0[j]
  for (j = 0; j < n; j++)
    v1[i] += A[i][j] * u1[j]
```

(b) All Outer Loops Fused

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    v0[i] += A[i][j] * u0[j]
    v1[i] += A[i][j] * u1[j]
```

(c) All Loops Fused

Figure 3.8: Three possible loop fusion options

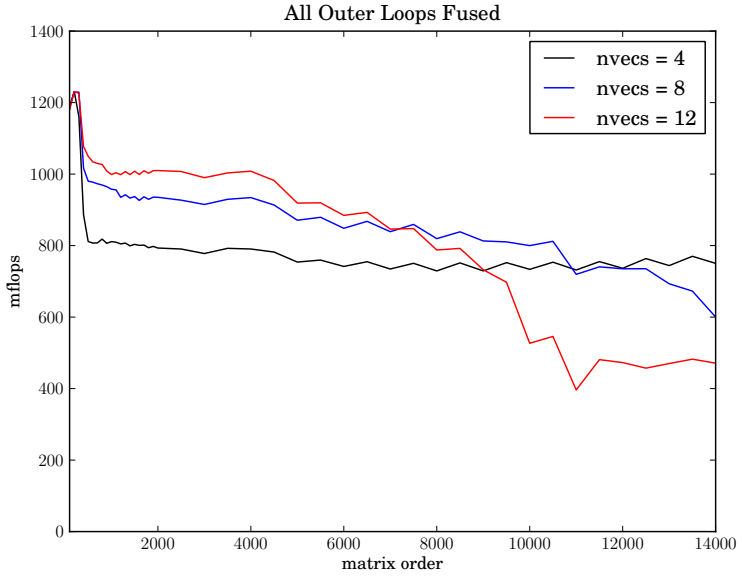


Figure 3.9: The performance of fusing only outer loops for various nvecs.

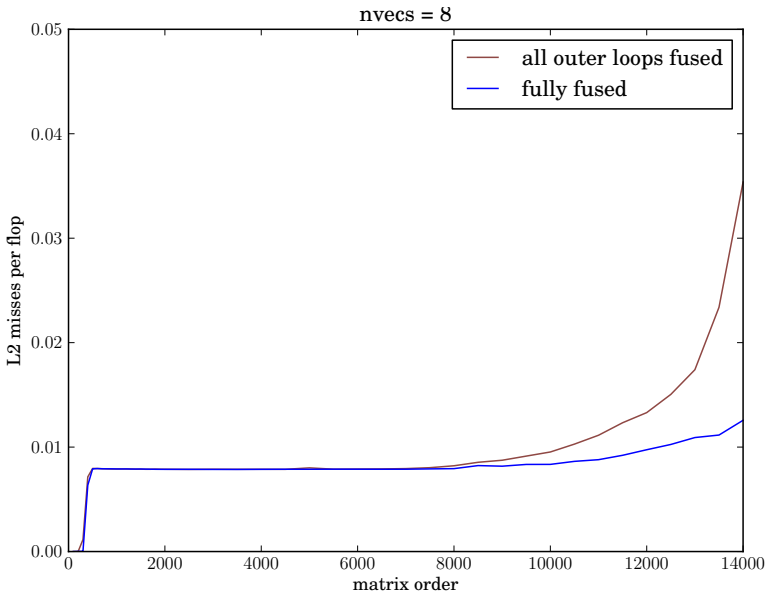


Figure 3.10: The memory effects of fusing loops for nvecs = 8.

3.4.1.2 Register Effects

Figure 3.11 shows the effects of fusing all loops of one to eight matrix-vector multiplies on the Opteron system for a fixed matrix size. The figure shows that fully fusing loops increases performance from one multiply through four. For five or more matrix-vector multiplies, the speed of the calculations decreases with each added operation. The slowdown is not a result of L1 and L2 cache and the TLB misses because the fully fused routine has the same number or fewer of these than the outerloop fused routine as shown in Figure 3.12. However, the assembly produced by icc shown in Figure 3.13 for these operations shows that the falloff in speed is a result of an increased number of `mov` instructions, each of which corresponds to an extra read from the L1 cache. The `mov` instructions occur because there are not enough registers to store the results of the inner most loop. When data accessed between successive iterations of the inner most loop cannot be stored in registers this is referred to as register spill. To prevent such register spill, the amount of fusion should be limited.

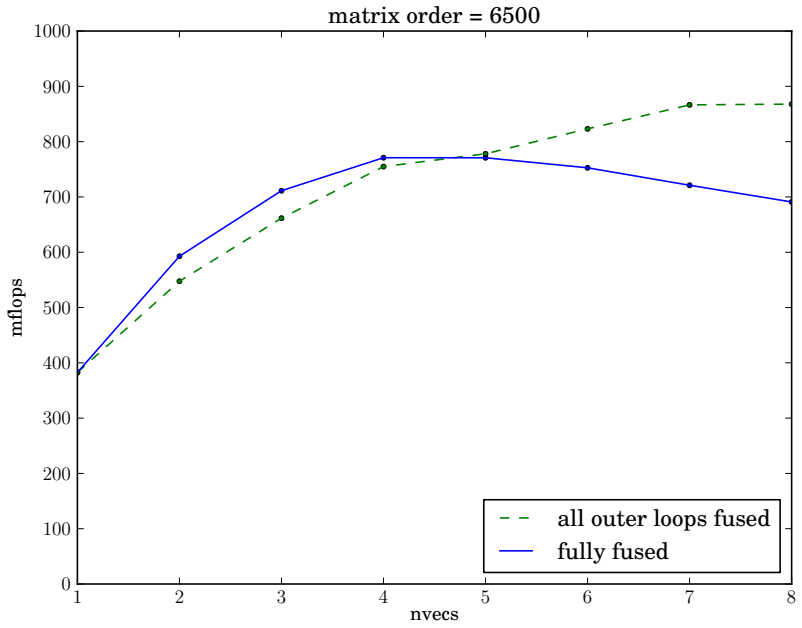
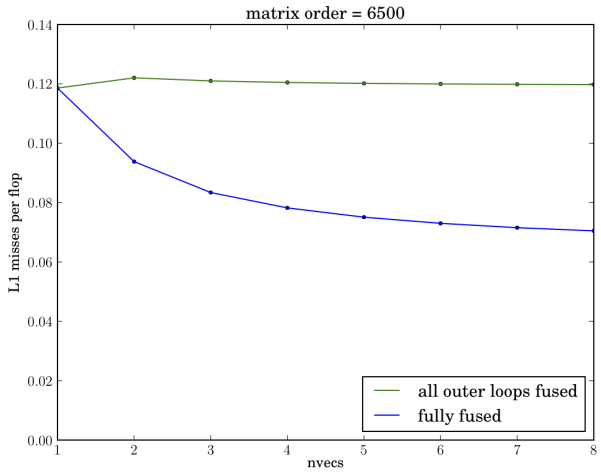
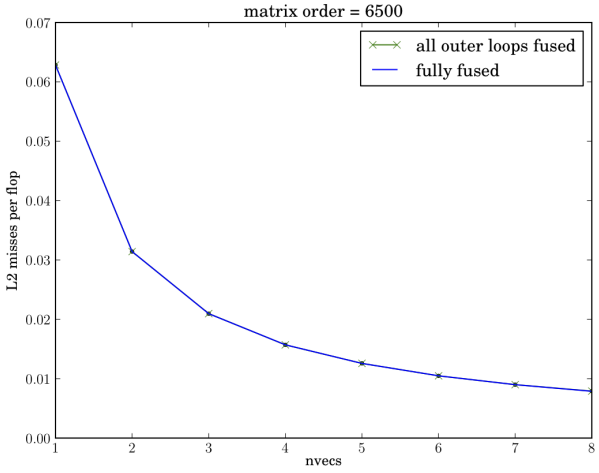


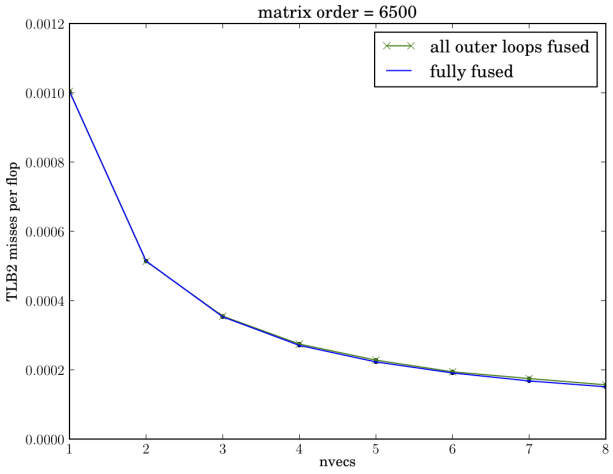
Figure 3.11: Performance of Fully Fusing



(a) L1 Cache



(b) L2 Cache



(c) TLB

Figure 3.12: Memory Effects of Fully Fusing

```
   ⋮          ⋮
faddl  (%edi,%eax,8)
fstpl  (%edi,%eax,8)
faddl  (%esi,%eax,8)
fstpl  (%esi,%eax,8)
```

(a) nvecs = 4

```
   ⋮          ⋮
faddl  (%esi,%eax,8)
fstpl  (%esi,%eax,8)
mov    0x4c(%esp),%edi
faddl  (%edi,%eax,8)
fstpl  (%edi,%eax,8)
```

(b) nvecs = 5

```
   ⋮          ⋮
mov    0x5c(%esp),%edi
faddl  (%edi,%eax,8)
fstpl  (%edi,%eax,8)
mov    0x54(%esp),%edi
faddl  (%edi,%eax,8)
fstpl  (%edi,%eax,8)
```

(c) nvecs = 6

Figure 3.13: Fully Fused Assembly

3.4.2 Suboptimal Memory Access Patterns

In the previous section of this chapter we describe how loop fusion can increase or slow the speed of calculations through its interactions with the memory sub-system. What is less apparent is that fusing loops can increase performance while not maximizing performance. For example, in the calculation of $AA^Tx = b$ in Figure 3.4, each matrix column $A[i]$ is accessed twice. If this vector is too large to fit in the level 1 cache, then the second read must come from the slower level 2 cache. In addition, when the vector is read in a second time from cache, the read is latency limited as the access is non-consecutive, because the sequence of reads jumps from the end of the vector to the beginning. The first read of the vector is consecutive because it follows the second read of the previous vector. Finally, the algorithm is not reading in new data from memory while performing $At = b$, which leaves the memory bus idle. As the movement of data over the memory bus is the limiting factor in routine performance, performing the calculation in this manner is inefficient.

3.5 Overcoming Bad Memory Effects of Fusion (Combining Fusion and Other Optimizations)

By combining loop fusion with other optimizations or limiting loop fusion to amounts that produce positive memory impacts, we can maximize the performance of calculations. In this section, we use the tuning techniques of cache blocking and software pipelining to overcome some of the negative effects of loop fusion. We also show that, by fusing the optimal amount, the performance of a routine can be increased.

3.5.1 Cache Blocking

For many calculations where loop fusion is profitable, more data must fit within caches for good performance to be achieved. In certain cases, the data do not fit within cache and performance of the routine suffers. Examples of this effect are shown in Section 3.4.1.1. To reduce the amount of data that must fit within cache for good performance, cache blocking can be used.

Figure 3.14 shows cache blocking applied to a fused implementation of $Ax = r$ and $A^T y = s$ shown in 3.1. By cache blocking only pieces of the vectors r and y are accessed by the inner most loop. When the block size is chosen to be small enough so these pieces remain in cache between iterations of the middle loop, performance gains result.

Figure 3.15 shows an example of performance gains attributable to cache blocking on the Hemisphere system. A corresponding reduction in memory reads is shown in 3.16. Both these figures show the calculation with the matrix divided into two blocks. Performance increases to within 10% of what it was without cache blocking and memory reads are reduced by approximately 50%. The gap in performance can be attributed to the fact that cache blocking creates high latency non-consecutive memory reads, which can also disrupt the memory prefetcher. Also, for smaller matrices cache blocking produces a slower algorithm than just fusing because it introduces non-consecutive memory reads and no reduction in memory reads.

```
blocksize = n/blocks
for  k = 1:blocks
  for  i = 1:n
    for  j = (blocks − 1) ∗ blocksize + 1:blocks ∗ blocksize
      r(j) = A(j, i) ∗ x(i)
      s(i) = A(j, i) ∗ y(j)
```

Figure 3.14: Cache Blocking $Ax = r, A^T y = s$

3.5.2 Software Pipelining

During the execution of the loop that computes $At = b$ for $AA^T x = b$, the memory bus is idle meaning that computing resources are not efficiently used. To keep data moving through the

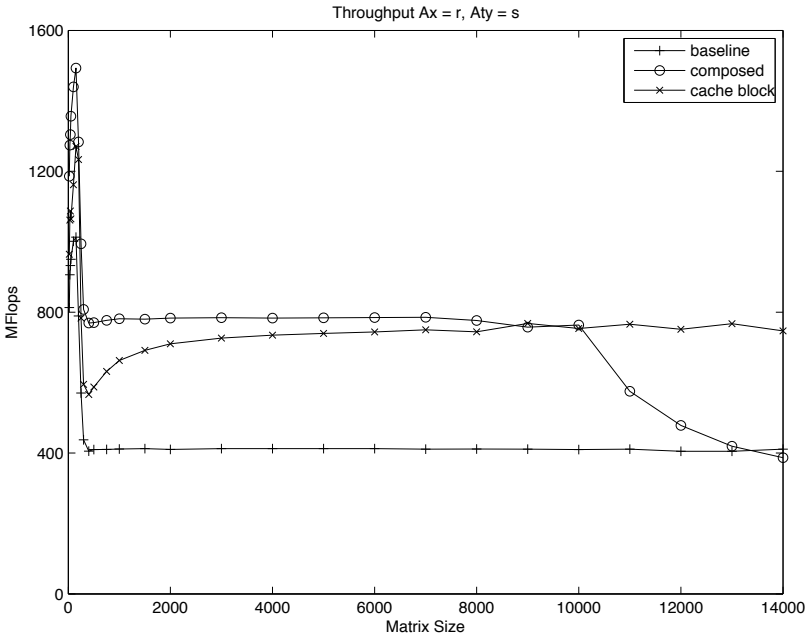


Figure 3.15: Performance with Cache Blocking

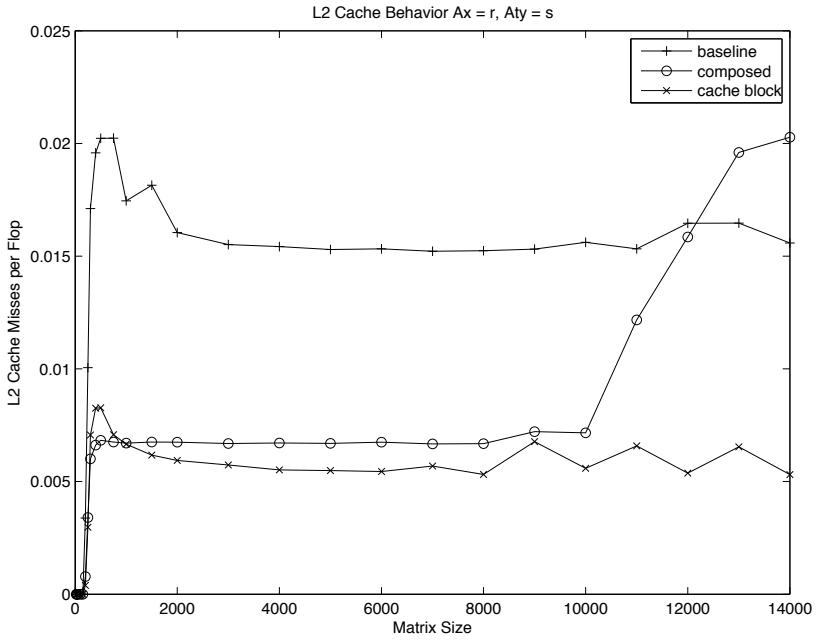


Figure 3.16: Memory Performance of Cache Blocking

bus, the computation can be rearranged using software pipelining. In order to do so, the $n-1$st iteration of the second inner loop is interleaved with the nth iteration of first inner loop. The first iteration of $A^T x = t$ and the last iteration of $At = b$ are performed in separate loops to complete the computation as shown in Figure 3.17.

```
for  j = 1:n
  t(1)+ = A(j,1) * x(j)
for  i = 2:n
  for  j = 1:n
    t(i)+ = A(j,i) * x(j)
    b(j)+ = A(j,i-1) * t(i-1)
for  j = 1:n
  b(j)+ = A(j,n) * t(n)
```

Figure 3.17: Software Pipelining $b = AA^T x$

Software pipelining results in a 30% speedup for matrix orders larger than 2000 on the Quadfather machine as shown in Figure 3.18. However, software pipelining does not affect the number of cache misses that occur during routine execution. The number of reads from each memory structure are the same as for the fused calculation without pipelining. The resulting speedup is a result of the memory bus' never being idle.

3.5.3 Not Fusing Too Much

Less outer loop fusion can increase performance by allowing more data to remain in cache and registers between loop iterations. In this section, we present examples of when reducing fusion allows data to remain in a cache or registers. In the case of register spill, reducing fusion is the best way to increase performance. For caches, though, other optimizations, such as tiling, are often a better choice.

Earlier we showed that fusing all the inner loops of six matrix-vector multiplies decreases performance when compared to fusing just the outer loops due to register spill. However, if we fuse the inner most loops in groups of three then performance is better than fusing all outer loops and all loops. When fused in groups of three there is no register spill, and some of the benifits of fusing

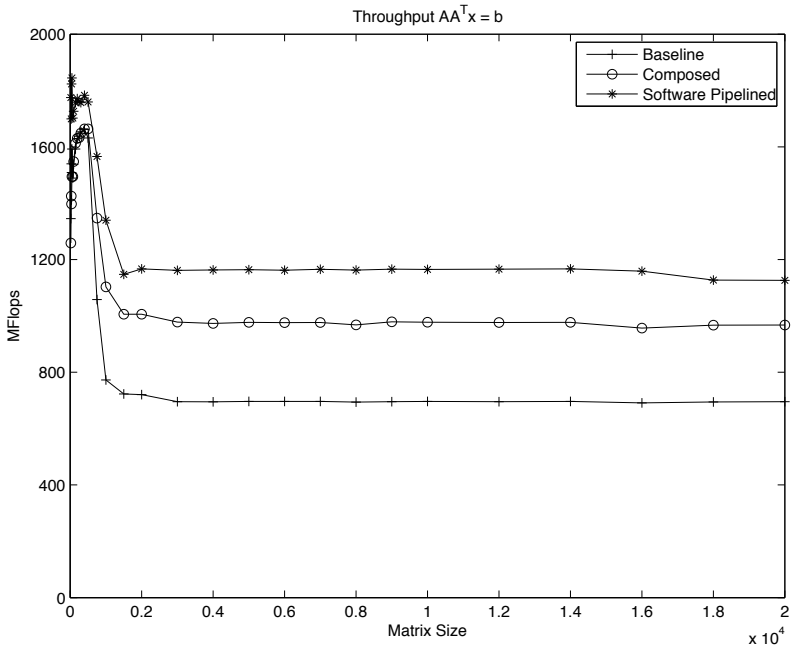


Figure 3.18: Performance with Software Pipelining

inner loops are realized.

When calculations become large or many are fused together, not fusing all the outer loops together can also increase the speed of a calculation. For example, when all outer loops are fused in Figure 3.9 the performance of the calculations begins to decrease at large matrix orders as level 2 cache misses start occuring. The more loops that are fused, the smaller the matrices for which these effects occur. The costs and benefits of not fusing, however, must be weighed against the benefits of combining another optimization such as cache blocking with loop fusion.

3.6 Search Complexity of Fusion Decisions

Finding the optimal amount of loop fusion is an NP complete problem [26]. Therefore, any consideration of loop fusion with other optimizations is also NP complete and the practicality of testing all possible loop fusion decisions is limited to a small number of loops. To create efficient code in a feasible amount of time, models [86] and/or heuristics [16] can be used to reduce the size of the search space. The risk in these cases is that local extrema can result in sub-optimal results.

Our approach to modeling fusion takes a different tack. We enumerate all possible options using the BTO compiler and then test only those that are predicted by our model to have the potential to be among the best routines. We explain how the model and compiler interact to efficiently produce high performing routines in Chapter 7.

In this section, we present an experiment that used the BTO compiler to show that certain fusion decisions never positively impact performance [62]. We plan to use these results to reduce our fusion search, which will reduce search times.

3.6.1 Similar Routine Performance

The enumeration of routines to be considered and the testing of those routines using hybrid search dominate the runtime of the BTO compiler. For many routines, multiple versions have near identical runtimes as well as model predictions of small runtime differences. For example, for the GESUMMV calculation shown in Figure 3.19, the compiler enumerates twelve possible versions

```
for i = 1:n                          //Loop 1
  for j = 1:n                        //Loop 2
    t1(i) = t1(i) + A(i,j) * x(j)
for i = 1:n                          //Loop 3
  t1(i) = α * t1(i)
for i = 1:n                          //Loop 4
  for j = 1:n                        //Loop 5
    t2(i) = t2(i) + B(i,j) * x(j)
for i = 1:n                          //Loop 6
  t2(i) = β * t2(i)
for i = 1:n                          //Loop 7
  y(i) = t1(i) + t2(i)
```

Figure 3.19: Unfused GESUMMV

with different amounts of fusion. The model produces predictions for the Core 2 system that all differ by less than 1%, and actual performance differences are less than 3% for the best and worst of these versions. Also, as shown in Figure 3.20, when we graph the actual and predicted runtimes for the GEMVER calculation for the Work system, we notice that many of the predicted and actual runtimes of routines are nearly identical.

A closer examination of these routines reveals that for many pairs of routines with near identical performance and predictions, the only difference between routines with near identical performance is the fusion of a vector operation with a matrix operation. If it were always the case that fusing a vector operation with matrix operations does not significantly improve performance then we would not need to enumerate and test these fusions in the BTO compiler.

3.6.2 Operations That Significantly Impact Performance

In this section, we show that fusing vector operations with matrix operations does not significantly impact performance. Throughout this section, fusion of a vector operation refers to the fusion of loops where each loop accesses the same vector. An example calculation with three sets of loops that contain vector operations and can be fused is the GESUMMV calculation shown with all loops unfused in Figure 3.19. Loops 1 and 2 can be fused with loops 4 and 5 to reduce the number of accesses to the vector x, where each element is accessed n times. Loops 3 and 6 can both be

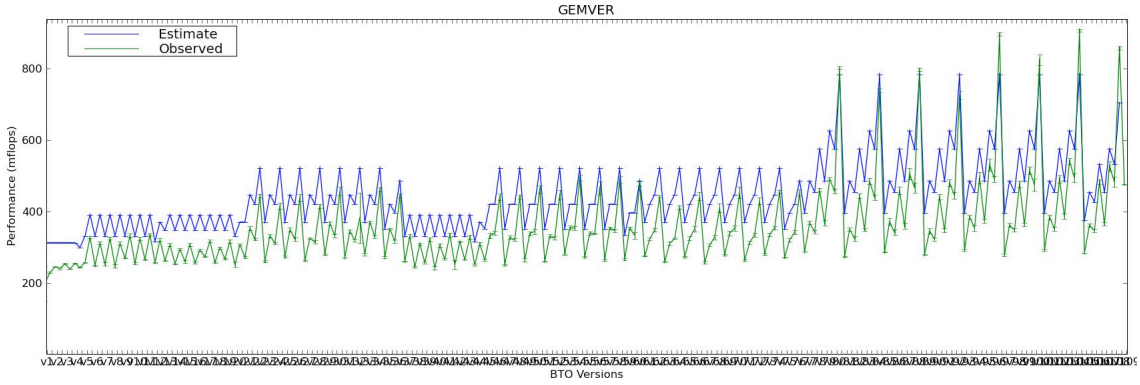


Figure 3.20: All Versions of GEMVER Actual and Predicted Performance

fused with loop 7 reducing the number of accesses to each element of the t_1 and t_2 vectors by one.

To determine whether fusing vector operations with matrix operations significantly impacts performance, we ran a series of tests. The test results were then analyzed to determine the significance of fusing vector operations. In this section, we first describe the environment, routines and methodology used to perform tests. We then present the results of these experiments including a statistical analysis of the results when needed.

3.6.2.1 Test Environment and Methodology

To determine the impact of fusing vector operations with matrix calculations, we ran the three calculations GEMVER, GESUMMV and DGEMVT on the Work, Nahalem, PowerPC and Opteron machines. All tests were compiled using gcc with the -O3 compiler flag turned on. The DGEMVT and GEMVER kernels were chosen from the updated Basic Linear Algebra Subprograms [15] and contain vector operations where the vector is accessed only once. The GESUMMV operation was chosen because it contains vector operations where the vector is accessed both once and multiple times. For DGEMVT, there are two sets of loops that can be fused that contain vector operations. For the GEMVER and GESUMMV calculations, there are respectively four and three sets of loops that containing vector operations that can be fused. All routines were chosen because they occur in important numerical linear algebra routines such as Householder bidiagonalization [55].

Table 3.3: Search Space of Routines

Routine Name	With Vector Operations	Without Vector Operations
DGEMVT	8	2
GEMVER	648	162
GESUMMV	12	3

Routines were run five times for each test of interest and performance differences less than 3% were considered small enough not to be significant. Any differences in the means of two routines being compared that was greater than 3% were subjected to statistical analysis to determine if the differences were statistically significant at a 95% confidence level. One directional Student's paired T-test [59] was used to compare the results.

Our null hypothesis for the T-test was that fusing vector operations never resulted in a statistically significant performance increase. Therefore, a one directional T=test was used because we want to identify when fusion improves performance. If fusion negatively impacts the performance of a routine in a statistically significant manner then the hypothesis is accepted. The hypothesis is considered true unless the p value from running the T-test is less than 0.05. A p value is the probability that the result of the test conducted occurred, assuming the null hypothesis is true. A p value below 0.05 indicates there is a statistically significant difference between the means of the two samples we compared.

3.6.2.2 Results and Analysis

The middle column of Table 3.3 shows the number of ways to fuse each routine with all vector operations considered. In all cases, we compared the performance of fusing and not fusing each operation by keeping all other fusion decisions the same and only changing the loop of interest. When a single pair of loops had a performance difference of more than 3%, we then used Student's paired T-test to determine if the differences were significant. The paired T-test was run for all pairs of routines that were identical except one routine fused a vector operation with a matrix operations and the other did not.

For the DGEMVT and GEMVER calculations, the fusion of vector operations never significantly impacted routine performance. On the Work, PowerPC and Nahalem machines, the performance differences were always less than 2%. On the Opteron, differences were larger and a paired t-test analysis, for the four pairs of interest, resulted in p values of 0.14 to 0.68. Therefore, there was not a statistically significant difference in runtime for any loop pair of interest at the 95% confidence level.

For the GESUMMV calculation on the Work, Nahalem and PowerPC, system, differences in performance were always less than 3% for all fusion possibilities. On the Opteron system, however, performance differences were greater than 3%. For vectors accessed once, the difference were not statistically significant with p values of 0.27 and 0.087. For vectors accessed many times performance differences of over 30% resulted as shown in Figure 3.21. The top line shows the performance of fusing all vectors accessed once, while the bottom line shows the performance of fusing all vector operations. The resulting speedups were statistically significant with a p value of 0.04349 for a matrix order of 3000.

From these experiments, we conclude that the fusion of vector operations where each element of the vector is accessed many times can significantly increase performance and must be considered when fusing loops. We also conclude that vector operations that only access elements once do not significantly impact performance when fused and can be removed from the search space.

3.6.3 Removing Vector Operations

For each routine we tested, there are two vector operations where all elements in the vector are accessed once. Removing the fusion of these operations with each other and with matrix operations from the space of considered routines results in a 75% reduction in the number of routines to be tested as shown in Table 3. For most routines, being able to eliminate a vector operation from the search space reduces the number of routines to be consider by approximately one half.

To perform this reduction within the compiler, we have developed two potential strategies. For each, we need to determine the relative cost of various operations. One option is to always fuse

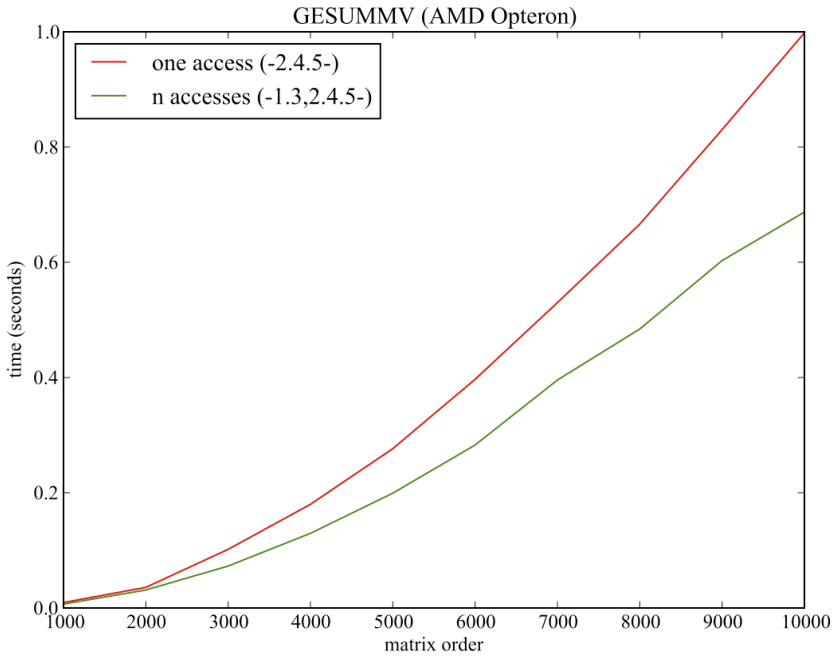


Figure 3.21: Runtime of fusing a vector operation for the GESUMMV calculation where the vector is accessed n times on an Opteron system.

vector operations when enumerating routines and then unfuse them when modeling and empirically testing their performance. Another option is to only fuse vector operations with matrix operations when fusing the vector operation enables the fusion of matrix operations.

3.7 Summary

Loop fusion is a powerful optimization that can dramatically reduce the runtime of linear algebra routines by reducing data movement. The impact of loop fusion on linear algebra routines has led to work on optimizing important routines highly, adding loop fusion capabilities to optimizing compilers and domain specific compilers. However, loop fusion can increase data movement and slow down routine performance if not carefully applied. Therefore, when fusing loops it is important to consider the underlying hardware and other optimizations being applied.

Chapter 4

Predicting Memory Traffic on a Single Processor

To predict the amount of data moving between each memory structure and the processor for fused linear algebra calculations, we developed a memory model. The model takes in a tree representation of calculations and uses reuse distances to determine from which memory structure each data structure is read. A reuse distance is a measure of how many unique data elements are accessed between accesses to a data element. The model also predicts how many times a data structure is read from each memory structure.

In this chapter, we first present the tradeoffs the memory model uses and explain the reasons for these tradeoffs. We then present the framework the model uses, including the data structures that store information about calculations and memory structures of a machine. Next, we explain how we predict misses for both caches and registers. Finally, we present how the model was validated, including a comparison of misses measured from hardware performance counters to model predictions.

4.1 Runtime and Accuracy Tradeoffs

To be useful within the BTO compiler, our model must be able to accurately identify the best routines in a practical amount of time. To meet this constraint, we restrict our model to calculating only the most distinguishing memory factors that impact the performance of fused linear algebra routines. Assumptions are made about memory structure features, data access patterns and cache state that decrease model runtime, but impact prediction accuracy. In this section, each tradeoff

is listed along with why we use it and the impact it has on model runtime and accuracy.

Consecutive data access patterns: Data movement through the memory hierarchy is most efficient if data elements stored next to each other in memory are read one after another. A consecutive access pattern is one that always reads neighboring data sequentially. For arrays, a consecutive access pattern means that column matrices are read a column at a time and row major matrices are read a row at a time. Also, within the columns and one dimensional arrays, indices are accessed sequentially, which means that the second element of an array is read immediately after the first element. When possible, linear algebra codes are designed to use consecutive access patterns. For example, the BTO compiler does so when it creates loops. Accounting for non-consecutive accesses would require adding latency and cache line size into the model, which would complicate it and only add small accuracy gains to predictions in a few cases. An example of where accuracy gains would occur by not assuming consecutive access patterns is in the summing of a row major and a column major matrix into the row major matrix where it is impossible to access both data structures sequentially. Elements stored consecutively in one matrix are stored the size of either a column or a row apart in the other matrix. Therefore, adjacent matrix elements results in non-consecutive accesses to one matrix.

Latency not modeled: When a consecutive access pattern is used, latency bound reads are rare, occurring only when all iterations of an inner loop access the same element of an array and then another loop is incremented changing the element of the array accessed. Also, out of order processors can hide some or all of the effects of latency. Therefore, exactly how much a non-consecutive memory access slows down program execution is impossible to know until runtime. To estimate the cost of latency bound reads on out of order processors would require knowing the size of the instruction window and whether the cache is blocking or not. When the cache is non-blocking, accurate estimates depend on knowing how many misses can be outstanding while the cache still services hits. Even with the ability to model latency bound reads, the benefit for routines with consecutive access patterns only occurs rarely. Due to the limited number of routines affected by latency and the added complexity of calculating hardware parameters, latency is left

out. The marginal benefit of including latency do not justify the significant computational cost.

TLBs and caches can be treated the same: The main differences between TLBs and caches are a TLB page addresses more data than a cache line holds, while a cache has many more lines than a TLB has pages. The latency penalty of an unanticipated TLB miss is typically an order of magnitude or more larger than an unanticipated cache miss. TLBs are usually fully associative and caches are typically set associative. As shown later, these differences are among the least significant factors in memory prediction and impact different versions of the same routine equally. Therefore, treating caches and TLBs the same reduces model complexity and increases model speed without significantly impacting the accuracy of our predictions.

Fully associative memory structures: For set associative caches and some TLBs, assuming memory structures are fully associative causes predictions to produce an abrupt increase in cache misses while actual misses follow a curved slope. Section 4.4.2 contains graphs showing the results of this assumption and a full explanation of why actual measured misses on the computer follow a curved slope. Since most TLBs and registers are usually fully associative structures, this assumption does not affect them.

Line size is equal to word size: By assuming that all cache lines and TLB pages are one data element, the model ignores data that are moved but not used. When reading data consecutively, most but not all data moved through the memory hierarchy are used. For example, all but the first and last cache lines and TLB pages are fully used, but the first and last lines and pages may not be fully used if the calculation starts or stops reading data in the middle of a line or page. However, the data movement not modeled in the first and last page is insignificant and ignoring it in our model has a negligible impact on accuracy. To determine how much data stored in a TLB page or cache line is moved but not used requires knowing where data will be stored in memory at runtime, which is impossible to predict. If strided accesses were introduced into the model then this assumption would need to be relaxed in order to account for the extra data movement caused by only using part of a cache line.

Warm cache assumed: A warm cache already contains data relevant to the routine being executed. In almost all programs, the calculations we model occur in the middle of a string of computations or are called multiple times. In these cases, most of the data that can stay within cache during a computation are there to begin with. Therefore, to mimic this behavior, we model the calculation by assuming it was the last routine executed by the processor.

All TLBs and caches use Least Recently Used replacement: For TLBs, the cost of a page fault is large enough and they occur infrequently enough that LRU replacement is typically used. For caches, a pseudo LRU strategy is often used in each set to reduce cost [52]. Therefore, assuming LRU is a close approximation of how actual memory structures operate in real hardware.

Reuse distance at the first element of an array is good for all: The reuse distance of a value in an array can change at different points within the array. In practice, the reuse distance changes on the order of one dimension of the array size, and arrays are equally likely to have increasing or decreasing reuse distance as calculations move further away from the start of the array. Sacrificing accuracy here allows for significant runtime speedups of the model while still accurately being able to distinguish between routines with large memory traffic differences as we show in Chapter 7.

Reads and Writes to Registers are Instantaneous: In most modern processors, accessing registers is immediate unless there is a dependency between multiple accesses. When a dependency does exist, often it does not delay execution because out of order execution hides it. Additionally, no experiments we have run indicate that accessing data in registers limits performance. Therefore, with few possible instances where accessing data from registers impacts performance, it is not worthwhile to model dependencies that occur as calculations execute on machines.

Arithmetic Costs Ignored: For memory bound calculations, the best way to compare their runtimes is data movement [3]. Since our model is designed for memory bound routines with large data sets, we ignore the cost of executing arithmetic within the processor. Ignoring arithmetic costs sacrifices accuracy for small problem sizes but increases our model's speed. In the next chapter, the impact of this assumption is seen in our inaccurate runtime predictions for small calculations.

4.2 Theoretical Framework (Equations)

To allow a general implementation of our model for those who want to use different tradeoffs, we develop a set of equations. These equations also allow us to verify that the implementation of our model is correct.

To define equations for the number of accesses to each memory structure, we establish several auxiliary notions. Let x range over memory structures in a machine. We write $prev(x)$ for the next smaller memory structure than x. The function $prev$ is undefined for the smallest memory structure (typically registers) and set equal to the number of load and store instructions completed during program execution. We use the symbol $\perp$ to represent the loads and stores. The amount of data a memory structure x can hold is written $size(x)$.

To represent all memory accesses within a loop L, we use a multiset of addresses $R(L)$. Each address occurs once in $R(L)$ for each time it is accessed in loop L (not including subloops). Let d range over memory addresses. We write $R(L)(d)$ for the number of occurrences of d in $R(L)$. We write $L_1 \leq L_2$ when L_1 is either the same loop as L_2 or nested somewhere within L_2. For a loop L, the working set of the loop, written $WS(L)$, is the number of unique data accesses in the loop (including sub-loops).

$$WS(L) = |\{d \mid 0 < \sum_{L' \leq L} R(L')(d)\}|. \tag{4.1}$$

The reuse distance of data element d in loop L, written $RD(d, L)$, is the number of unique data accesses between two accesses to d during the execution of loop L.

Now we define the hits to a memory structure x in loop L, written $H(x, L)$, and we define the accesses to x in loop L, written $A(x, L)$. These two multisets are mutually recursive and a (d) added to each applies the function to a specific input data element. However, the base case of A

does not rely on H while H always relies on A.

$$H(x,L)(d) = \begin{cases} A(x,L)(d) & \text{if } WS(L) \leq size(x) \text{ or} \\ & (RD(d) \leq size(x) \\ & \text{and } R(d,L) > 1) \\ 0 & \text{otherwise,} \end{cases} \tag{4.2}$$

$$A(x,L) = \begin{cases} R(L) & \text{if } prev(x) = \perp \\ A(prev(x),L) - H(prev(x),L) & \text{otherwise.} \end{cases} \tag{4.3}$$

The number of accesses to memory structure x in loop L (not counting sub-loops) is $|A(x,L)|$.

Equation 4.2 calculates how many data elements are read from each memory structure and not a larger structure based on reuse distances and working set size. A data element d that was not stored within a smaller memory structure is read from a memory structure if one of two conditions is met: The working set of the current loop is smaller than the memory structure's size or its reuse distance is smaller than the memory structure's size and it is not the first read to d. Otherwise, the read occurs from a larger memory structure.

Equation 4.3 totals up how many data elements pass through a memory structure assuming inclusivity among memory hierarchy levels. For the smallest memory structure, (usually registers), all are read from or transferred through it during execution. For all other memory structures, the amount of data that moves through it is the amount of data moved through the next smaller memory structure minus the amount of data read from the smaller memory structure and not larger structures.

4.3 Implementation

Our implementation of the model uses the equations in the previous section and the assumptions listed in Section 4.1 to predict memory traffic. The implementation takes as input an abstract syntax tree representing the calculation to be analyzed and a machine structure depicting

the system on which the calculation is going to be run. The model then estimates how much data will be read from each memory structure when the routine executes.

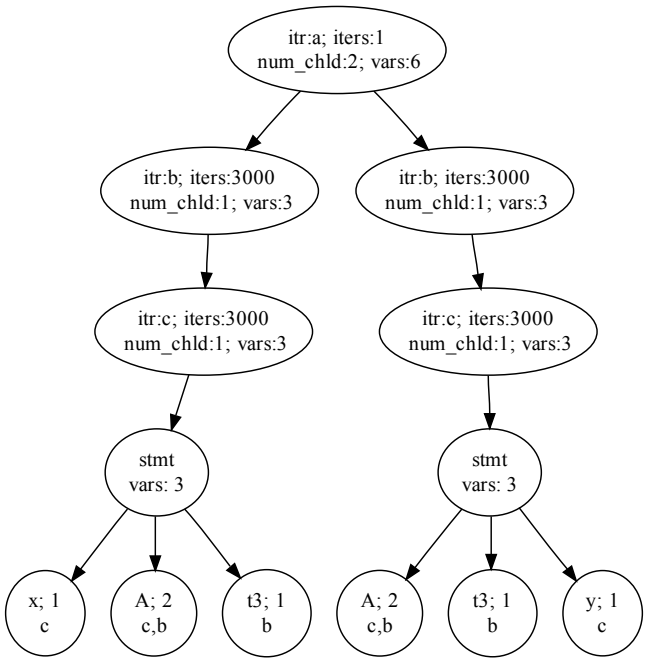
The abstract syntax tree, shown for both unfused and fused $b = AA^Tx$ routines in Figure 4.1, contains two element types: nodes and leaves. Nodes are loops and statements and leaves are variables. Each node contains the name of the variable that is incremented by the loop, and the number of iterations the loop is run, which is set to zero if it is a statement. Nodes also contain the number of variables that are accessed when the loop or statement is executed and pointers to those variables. The number of subloops and statements is stored along with pointers to them. For statements subloops is equal to zero and the pointers to other nodes are set to NULL. A loop also contains a pointer to the iterate that accesses the loop, which is set to NULL for a statement.

Variables are stored as leaves of the tree. If a variable appears multiple times in code, each instance of the variable is stored separately. Each leaf contains a variable's name, how many loop variables iterate the array and pointers to those variables. Additionally, a pointer to the number of misses to that variable is used to create an array with each value in the array corresponding to misses to one memory structure in the machine structure.

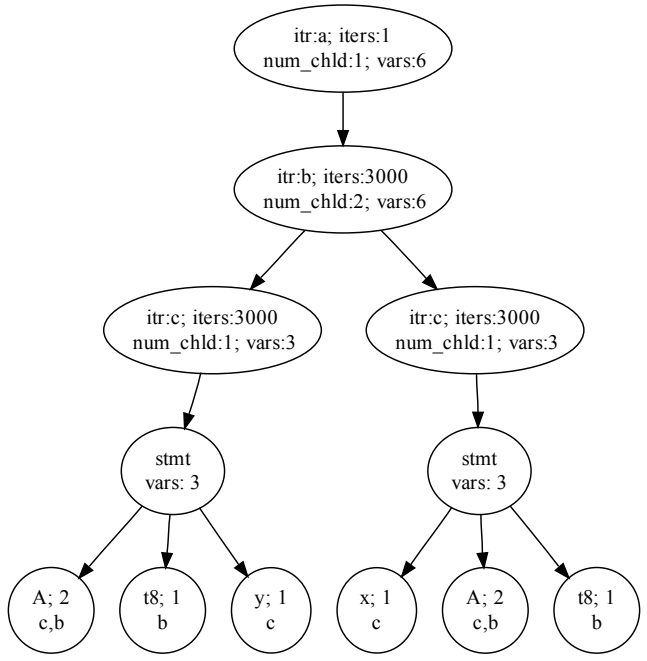
A machine is represented in a structure that contains its name, the number of memory structures it contains, and pointers to those memory structures. As shown in Figure 4.2 for a Core 2 system, each memory structure contains the amount of data it can hold or address, and a bandwidth, all expressed in bytes. The bandwidth represents how fast data can be moved from the next largest memory structure to the processor.

4.3.1 Cache and TLB Prediction

The memory prediction function starts at the outermost loop of the loop nest. Its first step is determining if the working set of a calculation or a loop is small enough to fit within a given cache. If the working set of a loop fits within a cache then the prediction function returns zero misses for that loop and cache. If the working set is not smaller than the cache then the function to calculate working set size is recursively called on each loop within the outer loop. The function



(a) Unfused



(b) Fused

Figure 4.1: The abstract syntax tree taken in by the model for $b = AA^T x$.

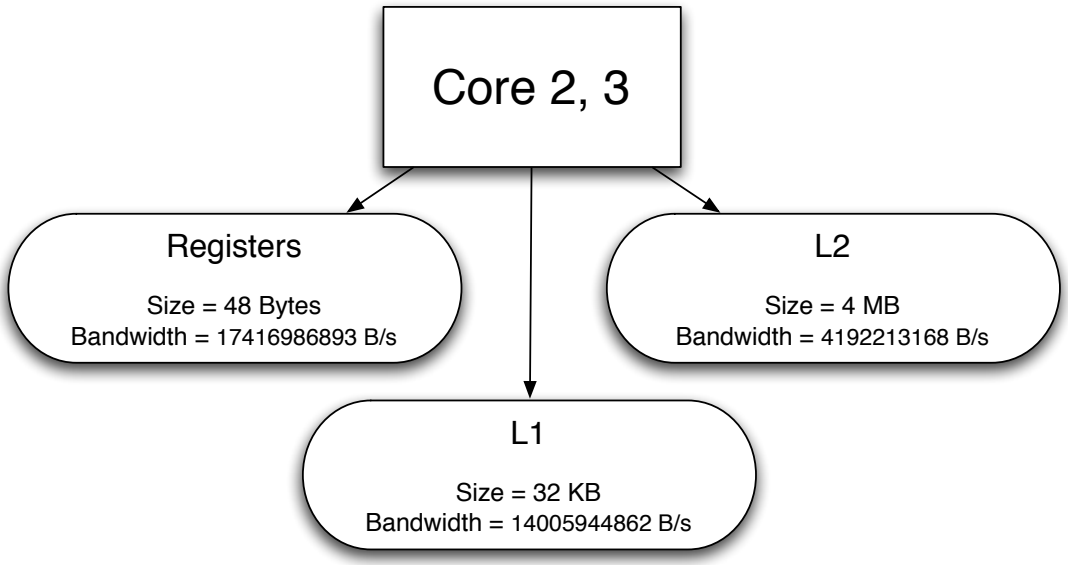


Figure 4.2: A machine structure representing a Core 2 machine.

is called until a working set smaller than the memory structure is found. If a working set smaller than the memory structure is now found misses are calculated for the innermost statement.

The first time the working set is not smaller than the memory structure size, misses are calculated for each instance of a variable. Each element of a data structure that is only accessed once by the current loop results in a miss to that memory structure. If elements of a data structure are accessed multiple times then reuse distances are calculated and used to determine if those reads occur from the current memory structure or a larger one. When the reuse distance is larger than the memory structure, each access to an element of the data structure results in a miss to the memory structure. When an inner loop contains misses, reuse distance is used for all variables without any misses in the inner loop to determine if misses occur to them in the current loop. For variables that already contain misses, the number of misses in the inner loop are multiplied by the number of iterations contained by the outer loop.

Misses to a data structure rather than hits are stored to reduce the number of memory structures represented in the machine structure by one. Removing one memory structure, usually main memory, is done because accessing registers is assumed to have no cost. Additionally, the implementation becomes cleaner because the bandwidth from the next largest memory structure to the processor is stored with that memory structure. Therefore, the number of misses and the bandwidth for the bus that missed data is moved over are stored together.

Reuse distances for arrays are calculated for the first array element only. Reuse distances are found by counting all unique data accesses that occur between two accesses to the first element of an array. When an element of an array is only accessed once, or for the first access to the array in a loop, the warm cache assumption is used in the calculation. The loop being modeled is assumed to have been executed twice in a row with the second execution being the one modeled. The reuse distance is then calculated from the last access to that array element in the first instance of the loop.

4.3.2 Register prediction

Registers must be treated differently in some respects from caches because the compiler is able to allocate them using additional information available to it. Therefore, we do not assume that they are allocated in a least recently used manner. To approximate how registers are allocated by the compiler, we use the following three heuristics. The first is that the iterate of an innermost loop is always stored in a register. The second is that variables accessed more than once within an inner loop are assumed to be stored in registers. The third is that values read from the same memory location within each loop iteration are likewise said to be stored in registers.

Also, all values stored in a register do not have cache misses while within the register even if their reuse distance is larger than the cache. Other than the heuristics presented in this section, registers are treated identically to caches.

4.4 Validation

To ensure the correctness and accuracy of our implementation, we validated our memory predictions using multiple techniques. To verify reuse distances and memory misses were being calculated properly the implementation was compared to the theoretical equations and code instrumented to track the number of reads and writes between successive accesses to the same data element. Additionally, predicted memory misses were compared to actual memory misses measured from hardware counters.

4.4.1 Comparison of Implementation to Equations and Instrumented Code

To ensure the correctness and accuracy of the implementation, we first compare the reuse distances computed when the analytic model is executed to those resulting from running instrumented code. The greatest possible difference between the two is twice the maximum number of variables in a statement (an array only counts once) times the wordsize (single or double) of the data structures being modeled. This difference results from the model's not enforcing the ordering

Variable	Instrumented RD	Model RD
a	81592	81600
x	1600	1608
t	81600	81600
a	81592	81600
t	81600	81600
b	1608	1608

Table 4.1: Instrumented and modeled reuse distances for unfused $b = AA^Tx$.

of variables within a statement. It is small compared to both cache size and reuse distance.

In Tables 4.1 and 4.2, we compare the reuse distances (RD) calculated by the model to those calculated by instrumenting code for the outer loop of matrix-vector multiples for the calculation of fused and unfused $b = AA^Tx$. The figure shows that the model predicts reuse to within twenty-four bytes for a matrix order of one hundred. Since all statements contain three elements and wordsize is eight bytes, the predictions are within our thirty-two byte tolerance. A small matrix order was used in the validation due to the long runtime of instrumented code.

Using the reuse distance calculations, we can calculate the number of misses to each memory structure that should occur per variable at each loop. Below, we compare the number misses per variable, produced by the model for unfused $b = AA^Tx$ to hand calculations for a 32 KB cache. Calculations are performed at the outer loop of the matrix-vector multiplies.

For the unfused routine the model predicts the following number of misses:

x: 800
A: 80000

Variable	Instrumented RD	Model RD
t	8	24
a	81600	81600
x	3192	3216
a	1600	1608
t	8	24
b	2408	2408

Table 4.2: Instrumented and modeled reuse distances for fused $b = AA^Tx$.

t: 800
A: 80000
t: 800
b: 800

Hand calculations of misses match the models estimates:

For both instances of A the reuse distance is 81,600. Since 81,600 > 32,768 cache size all 100 (rows) x 100 (columns) x 8 (wordsize) = 80,000 bytes of A are read from a larger cache.

For x and y their reuse distances (1608) are less than the cache's size. However, each is still read once from memory because between matrix-vector multiplies its reuse distance is larger than cache resulting in 800 bytes read.

Each instance of t in both matrix-vector multiplies has a reuse distance of 81,600 the first time it is accessed resulting in 800 bytes read from memory. However, subsequent accesses in the inner loop have a RD of 24, and, therefore, come from cache.

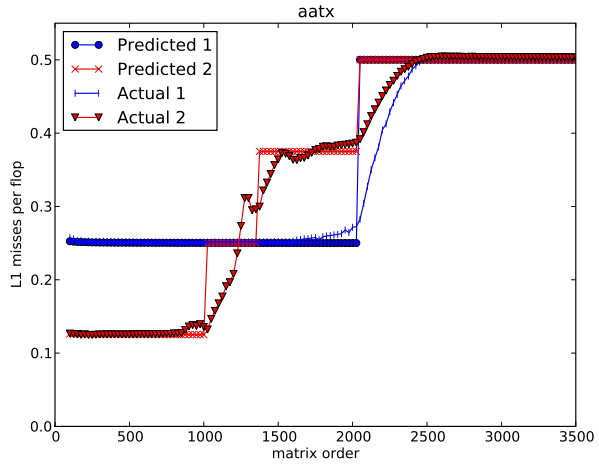
In all cases our model calculates the exact number of misses as our hand calculations.

4.4.2 Comparison of Predictions to Hardware Counters

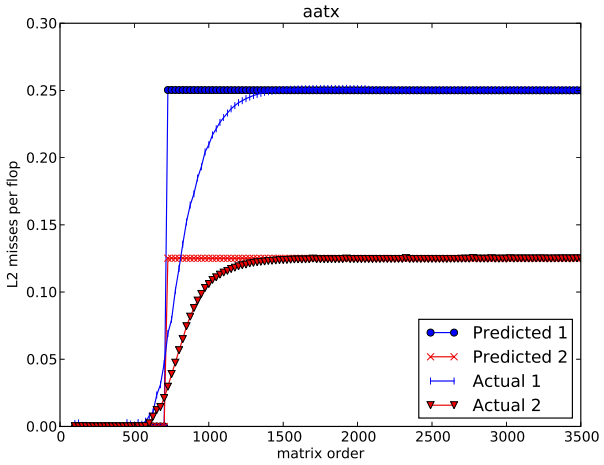
We also compare our model's memory read predictions to the actual number of reads measured by hardware performance counters. For each memory structure and loop, we divide the number of accesses by the memory structure's line or page size, written $LS(x)$, to obtain the number of cache lines or page table walks needed for that memory structure, written $LA(x)$:

$$LA(x) = \lceil A(x, L)/LS(x) \rceil. \tag{4.4}$$

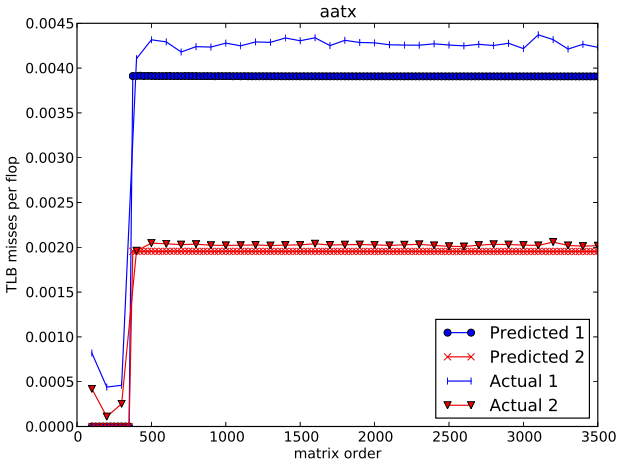
Figures 4.3 and 4.4 show the predicted and actual memory miss results normalized per floating point operation for the kernel $b = AA^T x$ on the Work and Opteron machines show in Table 3.1. Misses to a cache are equivalent to accesses from the next larger structure. Misses to the TLB result in page table walks. In each figure the actual 1 and predicted 1 lines show results with no fusion and the actual 2 and predicted 2 lines display the results for fully fused routines.



(a) L1

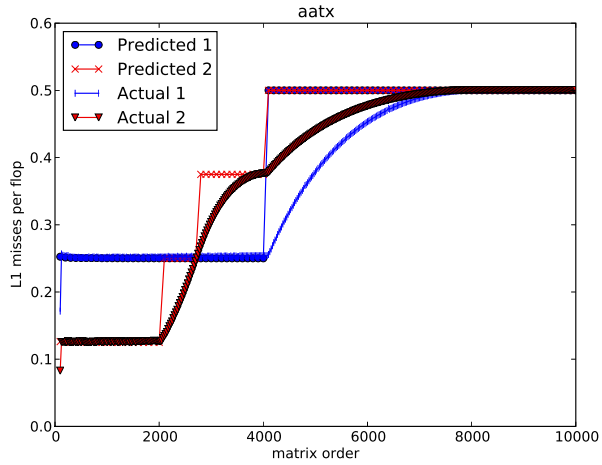


(b) L2

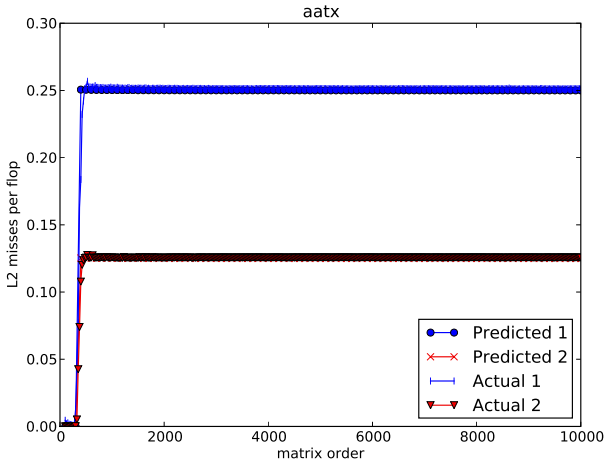


(c) TLB

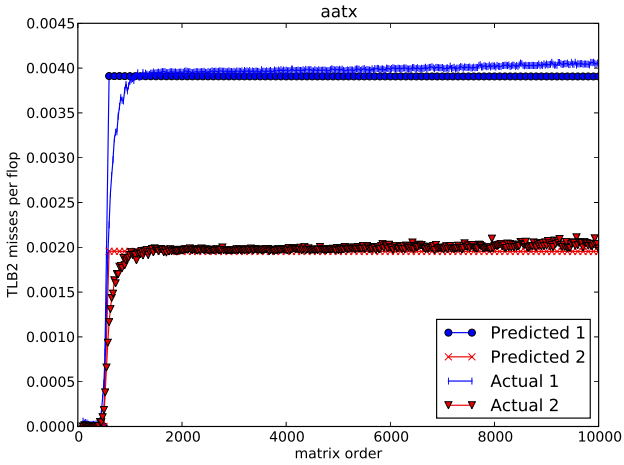
Figure 4.3: Predicted vs. actual memory misses of $b = AA^T x$ on the Work machine.



(a) L1



(b) L2



(c) TLB

Figure 4.4: Predicted vs. actual memory misses of $b = AA^Tx$ on the Opteron.

On both machines, the predicted misses for the L1 and L2 caches are accurate to within 1% except near cache boundaries. In those cases, conflict misses play an important role and expose the difference between the set associativity of the actual caches and the full associativity assumed by the model. Set associativity causes misses to begin sooner than predicted. Additionally, once the number of predicted misses spikes, fewer occur than predicted. The misses that happen before the predicted jump result in some cache sets filling before others and so causing conflicts. Actual misses then follow a curve as the matrix order increases due to progressively more sets having conflict misses. The curve of actual misses approaches the predicted number of misses from below because not all sets have conflict misses when a fully associative cache does have conflict misses.

The predicted misses for the TLB are accurate to within 10% for large matrices on the Work machine and within 2% on the Opteron except at TLB boundaries. On the Work machine the TLB is fully associative so conflict misses do not result and the prediction of where misses should begin to occur actually do line up. The Opteron's TLB is four way set associative resulting in a curve in the predictions for the same reason as the cache predictions.

Chapter 5

Runtime Prediction

The model uses a runtime prediction function to convert memory traffic estimates into a single value in seconds that allows the direct comparison of different implementations of the same routine. The function takes as input the estimated memory accesses to each loop determined by the memory prediction function presented in Chapter 4 and the usable bandwidth between each memory structure and the CPU. These inputs are then converted into runtime estimates.

In this chapter, we first describe how we automatically determine machine characteristics. Machine characteristics include number of registers, number of caches and their sizes, and the useable bandwidth between the processor, caches and memory. Then we present our serial runtime prediction function. Finally, we examine the validity of the runtime prediction function and present experiments showing its application to various problems.

5.1 Automatically Determining Machine Characteristics

To convert our memory traffic predictions to runtime predictions, the memory structure sizes and the bandwidth between these memory structures and the processor are needed. We determine these values through the use of an automated system, which increases the model's portability and ease of use. Automating this process eases the use of the model by making it unnecessary for users to determine the machine specifications and useable bandwidth of their system. Additionally, automation allows the model to be included with the BTO compiler without user input during the installation process.

In this section, we describe how we determine the number of caches on the system and their sizes. Also found at the same time is which processors share which caches and buses. We then explain how the number of registers on a system is determined. Finally, we present how the bandwidth from caches and memory to the processor is found.

5.1.1 How Many Caches and Their Sizes

To determine the number of caches and their sizes automatically we use Portable Hardware Locality (hwloc) [19]. Hwloc is a subproject of OpenMPI [43] that determines the memory hierarchy and processor layout of most modern computers. Information obtained includes how many cores and sockets a machine contains. Also determined is the layout of cores and caches on sockets. For example, hwolc discovers which cores share which caches. The single processor model uses the number of caches and their sizes, while the parallel model also uses the information about core, socket and cache layout. Hwloc works on operating systems as varied as Linux, Mac OSX, Microsoft Windows and Solaris. Its output is parsed and stored in a machine structure as described in Section 4.3 for serial predictions and a machine structure as describe in Section 6.2 for parallel calculations.

5.1.2 How Many Registers

To determine the number of registers accessible by the compiler on a system the following procedure is used:

A test code produces $n+1$ programs with bodies as shown in Figure 5.1 where j ranges from 0 to n. Each is compiled to assembly code and all lines containing 3456789 are counted and pulled out of the file for analysis. Lines that contain indirect addressing, which indicates the datum is read from memory are removed from the count. The program is run until three test codes in a row produce the same register count, which is then used to represent the number of registers on a machine. The procedure works in 32 bit and 64 bit mode for regular and vector registers.

Tests on Intel processors running both Linux and Mac OSX along with Opteron processors

```
register int a0;
register int a1;
       :
       :
register int an;
a0 = 03456789;
a1 = 13456789;
       :
       :
aj = j3456789;
a0 = a0 + a1 + ... + aj;
```

Figure 5.1: Code used to determine number of registers availible on a machine.

running Linux find the exact number of registers for 32 and 64 bit modes and regular and vector register choices.

5.1.3 Useable Bandwidth from Memory Structure to the Processor

To determine the amount of bandwidth available from a memory structure to the processor, we use the STREAM TRIAD benchmark [73]. STREAM is a widely accepted benchmark for determining bandwidths. TRIAD was chosen because, of the four stream benchmarks, it best approximates the expected mix of instructions for targeted applications. It requires two loads and a store to perform one multiplication and one addition, which is similar to the instruction and memory loads of many of the memory bound operations the model is designed to handle best.

The benchmark is run for each memory structure other than registers. For the smallest cache, the data size used is half of the cache size. For all other caches, the benchmark uses a data set that is the average of the size of the cache being profiled and the next smaller cache. To profile the bandwidth from main memory to the processor, we use a data set three times the largest cache size. The bandwidth for the smallest cache is stored in the machine structure with the registers, the bandwidth for the second smallest cache in the machine structure with the smallest cache and so on until the bandwidth from memory to the processor is stored in the machine structure with the largest cache. Therefore, each memory structure stores the bandwidth available to move data that does not fit in that memory structure.

5.2 Converting Memory Traffic Predictions to Runtime Predictions

To create a single value for predicted runtime from memory miss predictions, we use a runtime prediction function. The function takes in memory misses determined by the hardware model described in Chapter 4 and bandwidths discovered by using STREAM in the hardware profiling system presented in this chapter. Cost predictions can be used to compare the predicted performance of different versions of the same calculation or to estimate the runtime of a routine. In this section, we present equations that express the runtime prediction function and then describe the

implementation of the function.

5.2.1 Theoretical Equations

We express the conversion of memory traffic and machine characteristics to runtime predictions in a mathematical form. We continue to use the terms defined in Section 4.2 where x ranges over all memory structures and $A(x, L)$ is the number of accesses to the memory structure x for loop L. The runtime of the entire function is predicted by the following equations and the values calculated by the memory traffic prediction equations presented in section 4.2.

If L is an inner loop and $B(x)$ is the bandwidth between memory structure x and the processor, the *runtime* is computed as follows.

$$\mathit{runtime}(L) = \max\{A(x, L)/B(x) \mid \text{for all } x\}. \tag{5.1}$$

We use the largest value of $A(x, L)/B(x)$ over x because that represents the memory structure that is the bottleneck that limits performance for that loop.

We write $\mathit{child}(L)$ for each of the loops directly nested within loop L. If L is an outer loop, which is any loop with another loop inside of it, the *runtime* is computed as follows.

$$\mathit{runtime}(L) = \max\{\mathit{max}\{A(x, L)/B(x) \mid \text{for all } x\}, \sum_{c \in \mathit{child}(L)} \mathit{runtime}(c)\}. \tag{5.2}$$

When applied recursively from the root of the abstract syntax tree described in Section 4.3, Equation 5.2 produces a runtime cost estimate.

The sum of the inner loops' runtimes in this equation captures the impact of loops being bound by different memory structures, which can increase overall runtime. Once the sum's inner loops' runtime is found, it is compared to the predicted runtime of the outer loop. The greater of the two values is then used as the current loop's runtime.

5.2.2 Implementation

The runtime prediction function takes as input a machine structure as described above and the annotated tree containing memory predictions produced by the memory model presented in

section 4.3 and converts them into costs in seconds. The amount of time it takes to move data is found by dividing reads of it from a memory structure other than registers by the amount of usable bandwidth between that structure and the processor.

The first step in determining the cost of a calculation is a recursive traversal of the tree from root to leaves. At each node, the runtime prediction function is applied to the corresponding loop for each memory structure. The cost of the loop is the maximum of those contributions. The second step is a reverse traversal of the tree from leaves to root in which the cost of each parent node is set to be the greater of the cost returned from the runtime prediction function for it and the sum of the costs of its children. The cost of the root is the estimated cost of running the routine.

5.3 Validation and Accuracy

To validate the implementation of the runtime prediction algorithm and test its predictive accuracy, we performed the following tests. The values produced by the implementation of the algorithm were compared to hand calculations to make sure they were exactly the same. The model's runtime predictions were also compared to the measured runtimes of modeled routines. Additionally, the rank orderings of different versions produced by the model were compared to the rank orderings of actual runtimes of routines.

5.3.1 Validation of Implementation

The following procedure was used to verify that the implementation of the runtime prediction function produces the same prediction as the theoretical equations. The implementation of the memory model in Chapter 4 was set to output its miss predictions. These values were then used in hand calculations as demonstrated below for the fused and unfused calculation $b = AA^Tx$, which is described in more detail in Section 3.2. The results of the hand calculations were then compared to the results of running the runtime prediction function with the same input machine and memory miss predictions. This procedure was completed for fused and unfused $b = AA^Tx$, $r = Ax$ and $s = A^Ty$, and the GEMVER calculation found within the new BLAS standard [15] to ensure the

accurate implementation of the runtime prediction function.

Here we show an example of our validation of these calculations for unfused and fused $b = AA^Tx$ with a matrix order of 3000 on the Work machine. The bandwidths from each memory structure to the processor are shown in Figure 4.2.

The following charts show the memory miss predictions produced by the model:

unfused:

	L1	L2	Reg
Loop1:	288048000	144096000	288048000
Loop2:	144024000	72048000	144024000
Loop3:	144024000	72048000	144024000
Loop4:	144024000	72048000	144024000
Loop5:	144024000	72048000	144024000

‘

fused:

	L1	L2	Reg
Loop1:	288048000	72072000	288048000
Loop2:	288048000	72072000	288048000
Loop3:	144024000	72048000	144024000
Loop4:	144024000	24000	144024000

‘

The resulting runtime estimates from the cost function are 0.034372 seconds for the unfused routine and 0.0274692 seconds for the fused routine.

Hand calculations result in the same predictions as the cost function. For the unfused routine the runtimes of the loops are found as follows:

Loop 1 = max(288048000/14005944862, 144096000/4192213168, 288048000/17416986893) = 0.0343723
all other loops = (144024000/14005944862, 72048000/4192213168, 144024000/17416986893) = 0.0171861

‘

Since the sum of loops 2 and 3 equal loop 1 this is the cost of running that loop.

For the unfused routine the runtimes of the loops are found as follows:

Loops 1 and 2 = max(288048000/14005944862, 72048000/4192213168, 288048000/17416986893) = 0.0205661
Loop 3 = max(144024000/14005944862, 72048000/4192213168, 144024000/17416986893) = 0.0171861
Loop 4 = max(144024000/14005944862, 24000/4192213168, 144024000/17416986893) = 0.0102831

Since the sum of Loops 3 and 4 is larger than Loop 2's cost 0.0274692 is used as the cost of Loop 2. That cost is larger than Loop 1's cost and is used as the calculations cost.

In both cases, the runtimes from our hand calculations match with the cost function's predictions verifying its implementation.

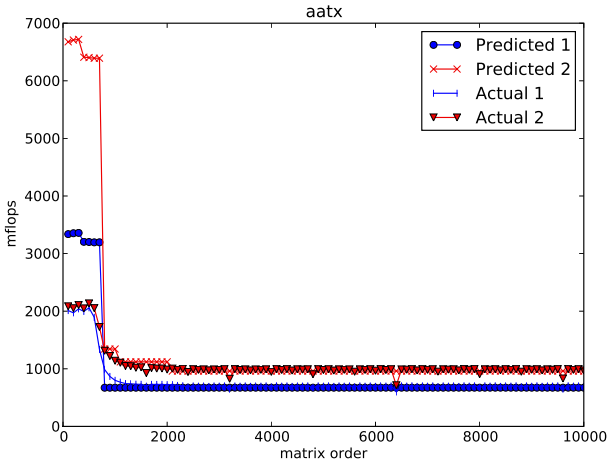
5.3.2 Runtime Prediction Accuracy

To show the usefulness and accuracy of the combined memory model and runtime prediction function, a series of tests was performed. Two criteria were used to judge the success of the combined system. How close runtime predictions are to actual runtimes measures the accuracy of predictions. Also, the difference in predicted runtimes between routines was compared against differences in their actua runtimes.

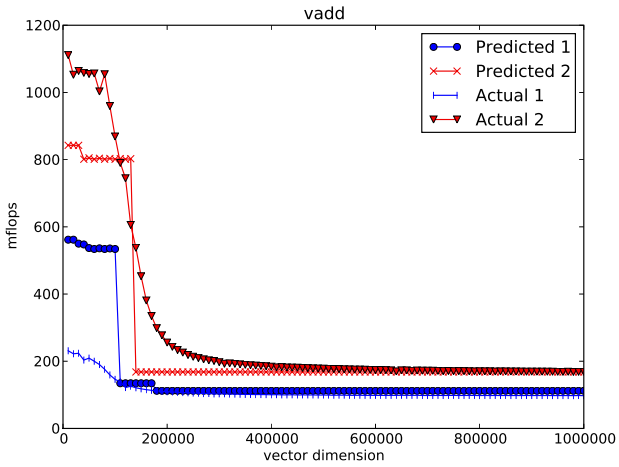
For using the model within the BTO compiler, this second criterion is especially important because accurate prediction is not as important as correctly identifying the best performing routines. Within BTO, it is also important that small differences between routines are not exaggerated and large difference are not minimized. Chapter 7 presents an analysis of the model's usefulness within BTO, while here we examine accuracy and correctness of ranking routines.

In Figures 5.2, 5.3 and 5.4, we compare the estimates produced by the runtime prediction function with actual runtimes of the compiler-generated codes on the Work and Opteron machines. The graphs in Figures 5.2 and 5.3 show the results for all versions produced by the compiler for three routines, $b = A^T Ax$, $x = w + y + z$ and $w = \alpha x + \beta y$. In each figure, the lines are numbered according to how much fusion was applied to that version of the kernel, with number 1 corresponding to no fusion and the largest number referring to the routine that is fused as much as possible.

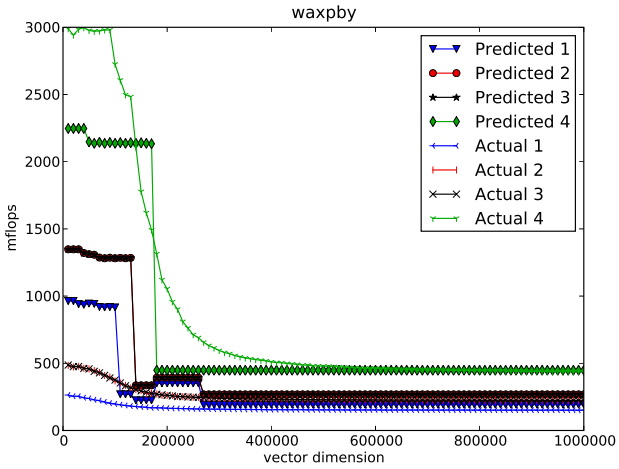
Figure 5.2 shows that our predictions for large matrices and vectors are accurate on the Work machine with predicted and actual performances nearly identical. Figure 5.3 shows that, on the Opteron, our predictions for large matrices overestimate actual performance on some routines by up to 25%. However, the relative ranking of the routines is accurate, meaning that within the compiler we are able to prune the search space using the model's predictions. For smaller



(a) AATX

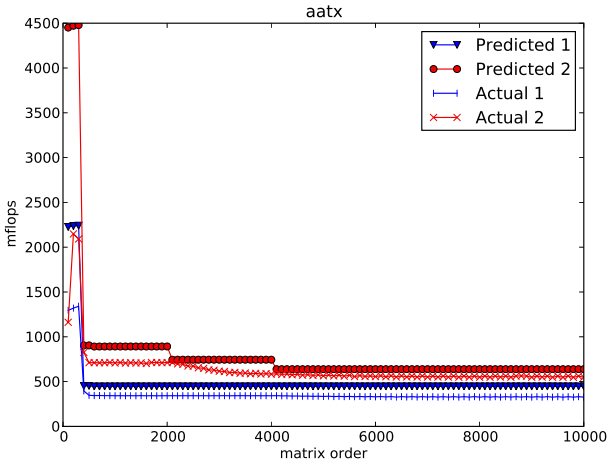


(b) VADD

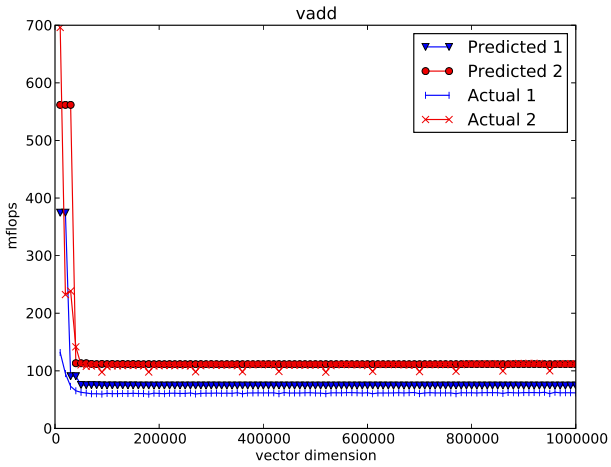


(c) WAXPBY

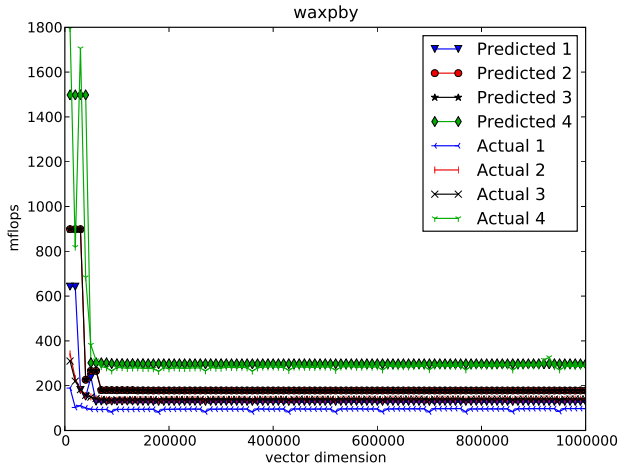
Figure 5.2: Predicted vs. actual runtime of three kernels on the Work machine.



(a) AATX



(b) VADD



(c) WAXPBY

Figure 5.3: Predicted vs. actual runtime of three kernels on the Opteron system.

matrices, our predictions are less accurate because the model does not take the cost of arithmetic or other computations into account, so the model overpredicts performance when computation is the bottleneck.

At cache boundaries, our predictions abruptly change while the compiler-produced versions follow smooth curves. The abrupt changes happen because the runtime prediction function uses the memory model's predictions. Therefore, jumps in performance in Figures 5.2 and 5.3 correspond to the the jumps in memory predictions in Figures 4.3 and 4.4.

Figure 5.4 compares our predictions to actual performance for the 648 versions of GEMVER produced by the BTO compiler at matrix order 3000 on the Work machine. It shows that, as the actual performance increases, the predicted performance, for the most part, increases as well. The model overpredicts performance in all cases, though this inaccuracy is less important than the performance difference between versions. On the Work machine the overpredictions occur because when kernels require temporary storage. When temporary storage is used the first write to a temporary array produces ten times the expected TLB misses. If we replace the TLB predictions with the actual number of TLB misses, the resulting costs are extremely accurate.

The last observation from Figures 5.2, 5.3 and 5.4 is that our runtime prediction function always ranks best kernel first.

In Section 3.4.1.2, we showed that data not being stored in registers significantly impacts routine performance and in Chapter 4 we show how we include them in our memory model. In Figure 5.5 we show how adding registers increases the predictive capabilities of the model. As seen in the figure the model including registers predicts a decrease in performance beginning at $nvecs = 5$, which is when five matrix-vector multiplies are computed. This drop off matches the behavior of the measured performance, while the model without registers predicts increased performance. At $nvecs = 5$, not all vectors in the inner loop of the fully fused calculation shown in Figure 3.8 remain in registers. The resulting growth in L1 cache reads causes performance to be bound on traffic from the L1 cache. Additionally, each increase of **nvecs** beyond five raises the ratio of L1 reads to computation and so reduces the performance predicted by the model.

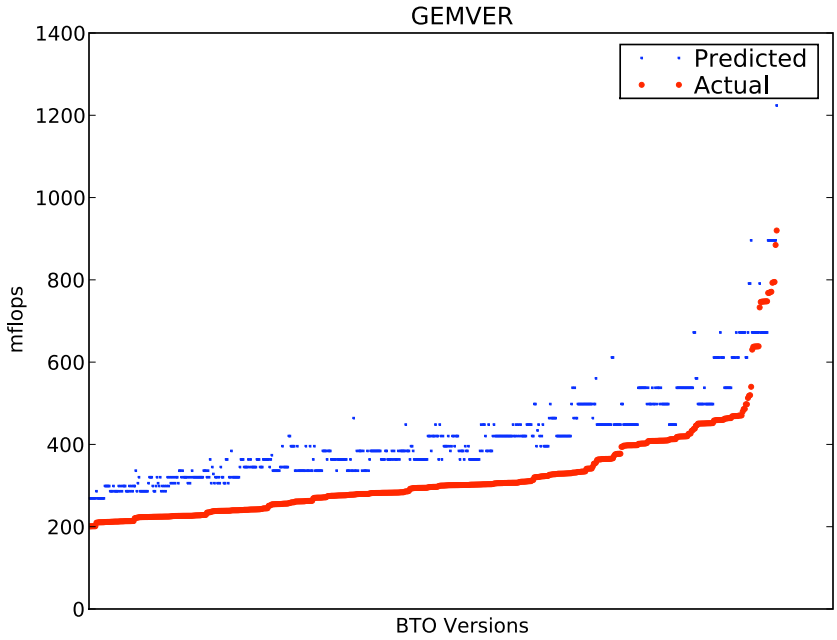


Figure 5.4: Predicted vs. actual runtime of the 648 versions of GEMVER produced by the BTO compiler on the Work machine.

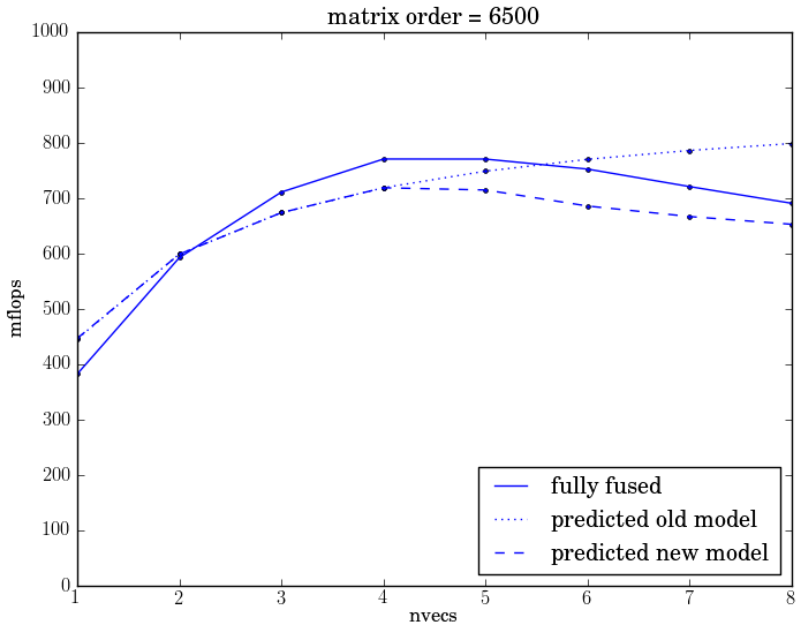


Figure 5.5: Accuracy of Model Predictions With and Without Registers

Chapter 6

Parallel Shared Memory Model

In order for our memory and cost predictions to be useful on modern machines, which often contain multiple sockets each with a multi-core processor, we add features relevant to parallel machines. To do so, we expand our serial model to account for additional hardware features such as multiple buses between main memory and processors that serial models do not incorporate. We also include which processors share caches and buses between the processors and how data and work is divided and distributed into our parallel model.

In this chapter and the following one, tests are run on some of the same machines listed in Table 3.1. In Table 6.1, we expand our description of the machines on which we run parallel tests by listing the number of processing units and memory structures they contain.

Name	**Sockets**	**Cores**	**L2 Caches**	**NUMA**
Work	1	4	2	No
Opteron	2	4	4	Yes

Table 6.1: Specifications of parallel test machines. All machines are homogenous and caches are shared by the same number of cores. The number of sockets is equal to the number of buses from memory to the processor in all cases.

In this chapter, we extend our serial memory model and runtime prediction function to SMP machines. First, we present how parallel machines change data and workload patterns. We then detail changes in implementation to our memory predictions for parallel machines. Finally, we

present changes to our cost function and introduce a second cost function that allows us to account for potential non-deterministic execution of certain routines on a parallel machine.

6.1 Parallel Machine and Execution Features

Additional processors, cores, buses, caches and registers in parallel machines increase the resources available to execute a program. However, shared caches and buses can impact runtimes in positive and negative ways through contention for resources and synergistic accesses. Also, division of data and workload between the cores and how a program is executed impacts performance. In this section, we explain how data division and program execution on parallel machines affect data movement and routine runtime.

6.1.1 Data and Workload Distribution

To take advantage of the additional processors, work must be divided among all of their cores. Two ways to partition the workload among cores are data parallelism and task parallelism. A data parallel approach splits the data being worked on among processing cores with each core executing the same instructions on different data. A task parallel approach allocates different operations to each core. The operations are then performed on the entire data set. A hybrid approach can also be used, with data divided among groups of cores that then split the tasks among themselves. How data and tasks are apportioned to cores impacts the amount of data stored in caches, reuse distances and data movement through buses.

For example, dividing the workload using a data parallel approach can result in extra data stored in shared caches and increased reuse distances. When computing a matrix-vector multiply, each core accesses a different part of the matrix. The data accessed by each core are stored in the shared cache, increasing the amount of data in the shared cache and the reuse distances of other data elements. Additionally, if the matrix is divided into groups of columns, each core produces a partial result of the whole product vector. The partial result occurs because each core only uses part of each row of the matrix. When performing a matrix-vector multiply on data divided by

columns, a column of the matrix is multiplied by single element of the multicand vector producing a partial resultant vector. These partial results are stored separately and summed, causing extra data to be resident in shared caches. The increased data stored in caches can produce additional misses, leading to increased reads from slower memory structures.

Another potential pitfall of data parallelism is contention for shared memory buses. If two cores are to read different data at the same time that data needs to move over the same memory bus. There are two possible ways the data can move over the bus, either one core stalls while waiting for the other core's read to complete or the reads are interleaved with each processor receiving data at half the transfer rate of the bus. However, there are no extra data in cache or bus contention if the calculation is performed on two cores using a task parallel division of work. With one core performing the multiply and the other the add, each accesses the same part of the matrix requiring no partial results and not causing bus contention.

Splitting data using task parallelism results in different potential pitfalls. If cores performing tasks on the same data each read that data through different memory buses then there can be an increase in total data movement as compared to serial and data parallel routines. Additionally, if the two cores become out of sync, such that data do not stay within shared caches between when two tasks access them, extra memory traffic can result. Finally, there also is added overhead needed to enforce dependencies between tasks. Data parallel divisions avoid both these issues, though cores finishing their workload later than others can lead to wasted clock cycles and memory bandwidth.

Both forms of parallel computation can increase or decrease data movement compared to serial execution. When used together, their strengths can be combined and weaknesses hidden. For example a calculation can be implemented using task parallelism when cores share caches and memory buses and data parallelism when cores do not share a bus. The combined approach can reduce contention on that memory bus, the amount of data stored in the shared cache and data movement. This hybrid approach can be the most effective way to perform a calculation. However, our parallel memory model is currently limited to modeling data parallelism because currently BTO does not create task parallel routines.

6.1.2 Routine Execution

Due to non-deterministic execution patterns, acounting for how a routine can be run is necessary on parallel machines. Different execution patterns result in verying amounts of contention for hardware resources occur among cores. In certain cases, one execution pattern might sometimes leave the bus that limits data movement idle while a different pattern does not. When non-deterministic execution occurs, the runtime of the calculation can vary depending on which execution pattern occurs during runtime. In other cases, both execution patterns can result in identical performance. For example when neither execution pattern leaves the bus that limits data traffic idle more than the other.

An example of a routine where contention may or may not occur depending on how a routine is executed is the fused kernel $b = AA^Tx$ shown on the right side of Figure 3.4. For this routine, we present two execution paths, which are demonstrations of possible performance execution, and use them to develop performance bounds. In actuality, a processor can switch between these execution patterns unless the code is written to enforce one or the other. We assume that two columns of A and all vectors fit within cache. For our examples, we run the calculation on two cores of a parallel machine that share a cache and divide the data by performing half of the outer loop iterations on each core.

For the first execution pattern, each core executes the same inner loop, $t = A^T * x$ at the same time accessing different columns of A that are read from memory at the same time thus creating bus contention. Then each core executes the second inner loop $b = A * t$. Since the data of A accessed by the second inner loop are in cache, there is no bus contention because no data are read from memory.

For the second execution pattern, the two cores become out of sync. While one core executes $t = A^T * x$ the other core executes $b = A * t$. Only the core performing $t = A^T * x$ core is reading data from main memory since the core performing $b = A * t$ is reading data from cache. Then the cores switch which calculation they are performing, resulting in no bus contention since only one

core is ever reading data from memory at a time.

For our machine with a single shared cache, if moving data from memory is the limiting factor for routine performance, then the first algorithm executes more slowly than the second. The disparity in speed is the amount of time it takes to execute $b = At$. The time difference occurs because the second algorithm hides the cost of performing $b = At$ by executing it on one core while $t = A^T x$ reads data from memory and runs on the other core. However, if data movement from cache limits performance, the runtime of both algorithms is the same since calculating both parts of $b = At$ at the same time does not create bus contention because cores executing the routine use separate buses to access cache. When and by how much execution patterns impact a routine's runtime is both machine and routines dependent.

6.2 Modeling Parallel Memory Traffic

To add parallel capabilities to our model, we made a series of changes. First we added more information to our abstract syntax tree (AST) and machine structure. An additional field is appended to all variables in the AST to mark those that store partial sums once per running thread. We also use this field to indicate data structures accessed from more than one location by different threads.

Machine structures were changed to store a hierarchy of structures. Figure 6.1 shows and example of the new machine structure for a dual socket Clovertown machine shown in Figure 2.1. The added fields of cores, threads and sockets each express how many processing units access that memory structure directly. At the top level in this figure is the machine level. The second level is the socket level for this machine. The third level represents the two cores that share the level 2 cache. The lowest level represents the individual cores, level 1 caches and registers. For homogenous systems or heterogenous systems with homogenous components, identical memory structures are compressed to a single representation. A new field called compressed is added and the number of memory structures represented by the representation is stored in the compressed field. Therefore, all memory structures stored at a given level of the hierarchy are accessed by the same number of

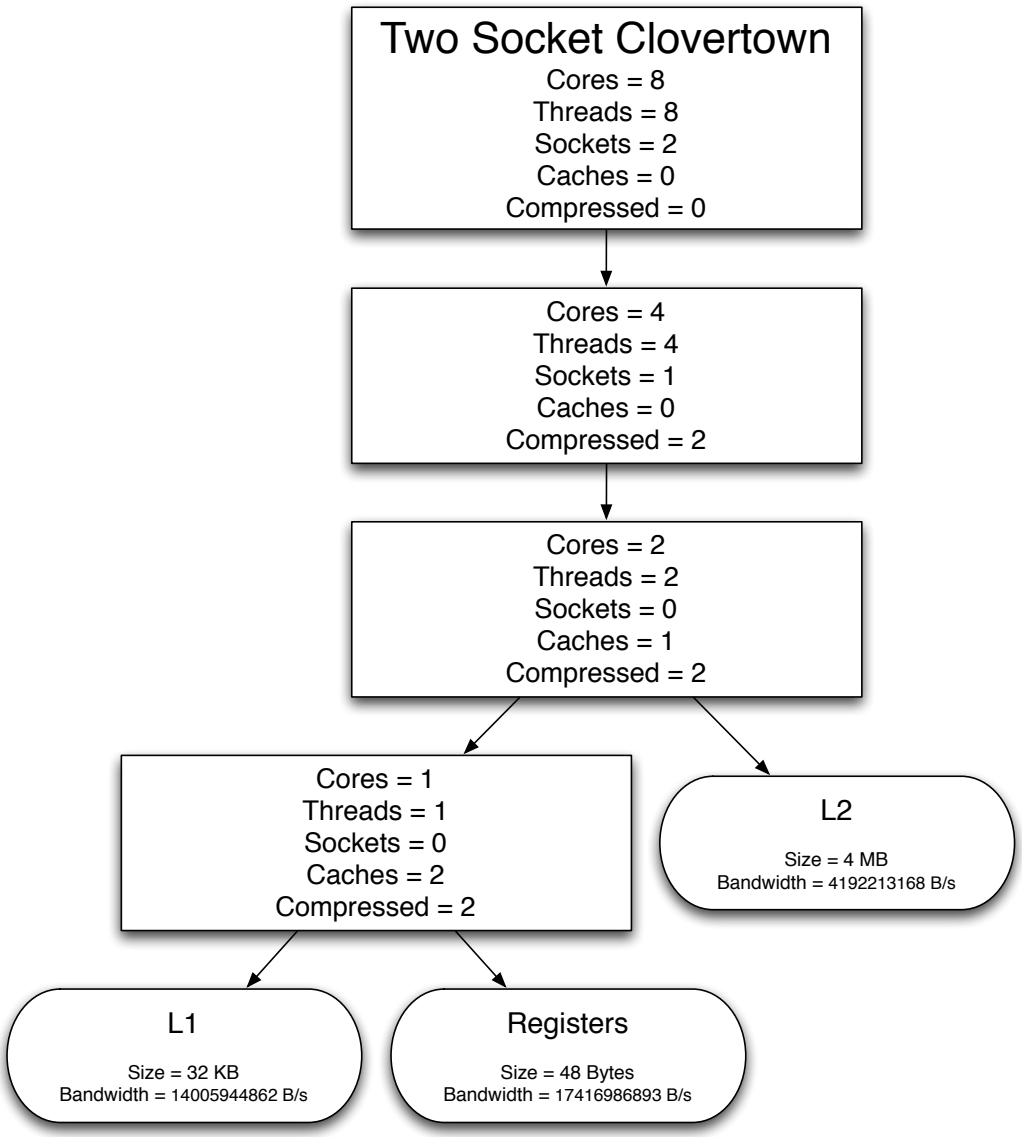


Figure 6.1: Parallel machine storage example.

cores, which is stored at that level.

Buses are represented implicitly with the number of buses from memory equal to the number of compressed structures in the second level of the hierarchy. The number of buses from all caches at a given level is found by dividing the total number of cores by the number in the next lower level of the hierarchy. The number of cores, threads and sockets at a level of the hierarchy represents how many of these hardware structures share all caches at that level.

Currently our model is only designed to work on homogenous machines, but all changes made to parallel system representation are forward looking towards the advent of generating kernels for heterogenous computing environments. In this section, we present our memory prediction algorithms for shared memory parallel machines that use these modified data structures. Implementation details are explained and then followed by analysis of the accuracy of our model's predictions. Since the theoretical equations for memory predictions presented in Section 4.2 do not change from the serial case, we do not present them here.

6.2.1 Parallel Memory Prediction

To predict the amount of data movement on a parallel machine, we must adjust reuse distance calculations to account for extra variables stored in caches, such as partial results and variables used by other cores that share a given cache. These additional data can increase reuse distances in shared caches when multiple cores access different parts of a shared data structure. Also increasing reuse distances are partial results that must be stored and accessed by different cores. If a partial result is reused by its core, but is accessed fully before the reuse occurs, then the reuse distance of each partial result is made larger by the number of partial results times the sizes of the partial results. In our model, we adjust our reuse distance calculations to account for these factors.

Whenever a partial result is updated between accesses to any data structure, the reuse distance of the data structure increases by the size of the partial result times the number of instances of the partial result stored in the memory structure of interest. The same adjustment is applied to all data structures divided among cores and accessed in different places by the cores. Calculating

reuse distances in this way accounts for both shared caches and the extra data stored in them when executing a calculation in parallel. The result is that reuse distances for the same variable can differ for various memory structures due to the number of cores that access that structure and, therefore, must be calculated per structure. Then to determine the memory structure from which reads to each data structure occur, reuse distances are compared to the memory structure's size.

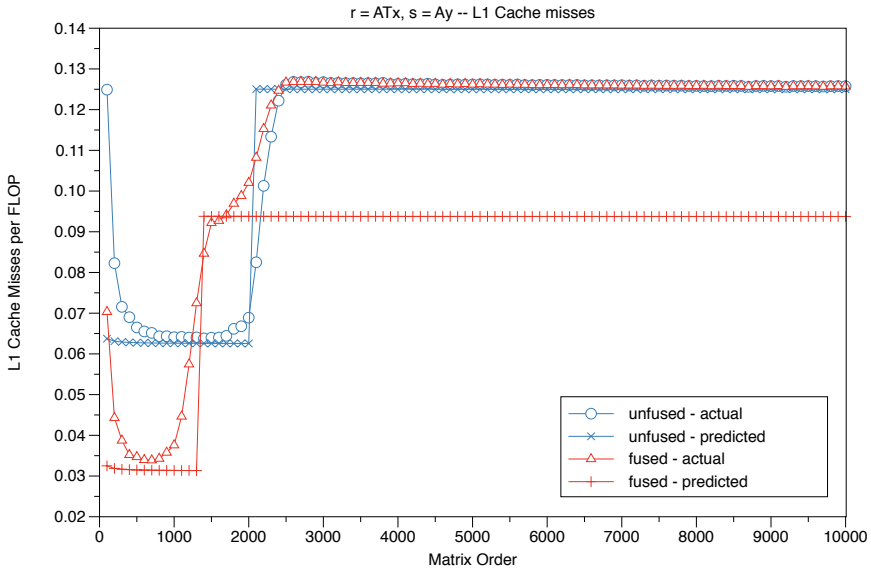
6.2.2 Prediction Results

The first step in evaluating the accuracy of our parallel memory model was comparing memory miss predictions to the actual number of memory misses observed by measuring hardware counters. In Figures 6.2 and 6.3, we compare the actual and predicted L1 and L2 misses for the Intel Core 2 Duo machine. Results are normalized per floating-point operation to aid in presentation.

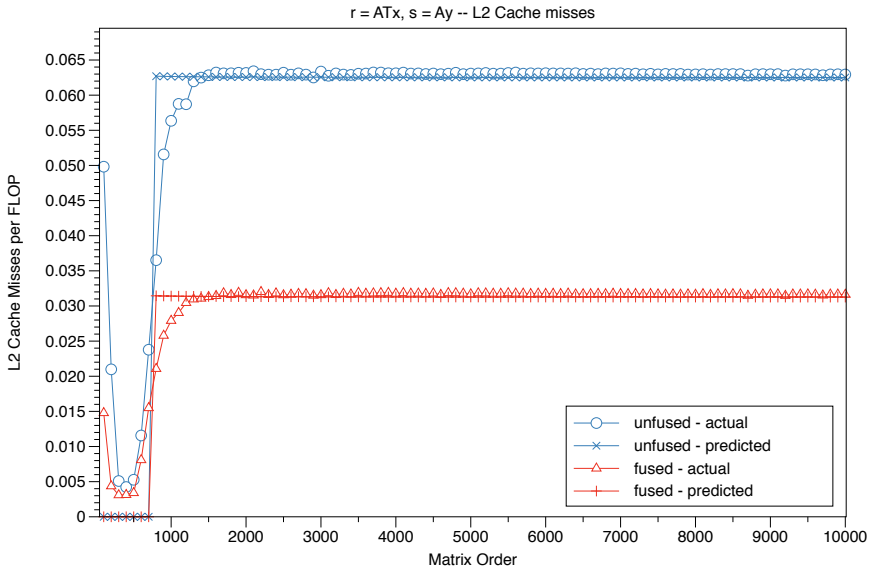
For small orders, overhead can cause our predictions to be inaccurate. Also, at cache boundaries, we abruptly predict increased misses due to assuming a fully associative cache while actual misses increase in a curve as explained in Section 4.1. Otherwise, for both calculations, our predictions are accurate other than in two noteworthy cases. For $b = AA^Tx$, the unfused calculations' L1 cache miss predictions are inaccurate for matrix orders of approximately 2000-4500. This probably occurs because the processor fits most of the x and t vectors into cache by not including a column of A. We are working on testing this explanation. The other inaccuracy is in the L1 cache miss prediction of the fused calculation of $r = A^Tx, s = Ay$. For large orders, our model predicts only about 75% of the actual misses. As for our previous inaccuracy, we assume the hardware cache strategy is causing the discrepancy.

6.3 Parallel Runtime Prediction

In this section, we explain how to convert parallel memory traffic predictions to runtime predictions. We present two functions for converting predictions to runtimes that provide best and worst case predictions when nondeterminism is possible. The section is organized as follows. First, we present theoretical equations for the parallel case. A proof of why the equations presented

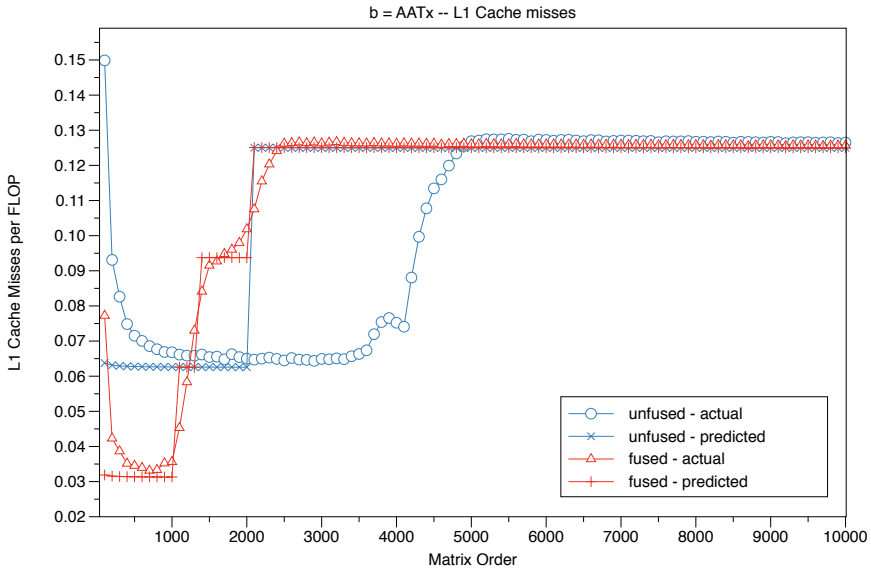


(a) L1 Misses

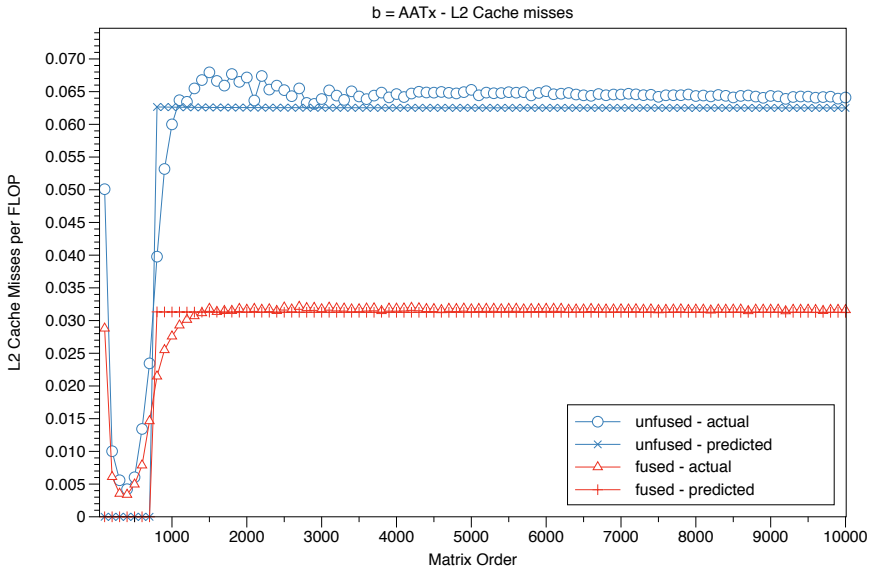


(b) L2 Misses

Figure 6.2: Predicted and actual cache misses for $r = A^T x, s = Ay$.



(a) L1 Misses



(b) L2 Misses

Figure 6.3: Predicted and actual cache misses for $b = AA^T x$.

represent the best and worst cases follows. We then describe the implementation of the model. Finally, we demonstrate the accuracy of our performance predictions.

6.3.1 Theoretical Expression

To provide a theoretical framework that is implementation independent we developed equations to express our parallel runtime predictions. These two equations are used to represent the worst case and best case execution scenarios described in Section 6.1.2.

The worst case is expressed by equations 6.1 and 6.2 which are modifications of the equations 5.1 and 5.2 for the serial case presented in Section 5.2.1. Equation 6.1 is applied to the innermost loops and equation 6.2 is applied to all other loops. To the serial equations, we add the term $Bus(x)$, which is the number of buses that all memory structures x use to access a processor. This added term expresses the additional bandwidth available due to parallel execution.

$$runtime\_inner(L) = \max\{A(x,L)/(B(x) * Bus(x)) \mid \text{for all } x\}. \tag{6.1}$$

$$runtime\_outer(L) = \max\{max\{A(x,L)/(B(x) * Bus(x)) \mid \text{for all } x\}, \sum_{c \in child(L)} runtime(c)\}. \tag{6.2}$$

The best case scenario assumes that all data movement not from the limiting memory structure is overlapped with data movement from the memory structure that limits performance. Therefore, the memory structure that bounds performance at the outermost loop level is the memory structure that limits performance. The limiting memory structure is found by applying equation 6.1 to all memory structures for the outermost loop. The runtime returned is the routine's performance estimate.

There is one exception to this use. We define $runtest(L)$ as the result of applying equation 6.2 to loop L. When the inequality in equation 6.3 is true for the outermost loop, data movement is not overlapped for memory structures closer to the processor.

$$runtime(L) * processors/Bus(x) < runtest(L) \tag{6.3}$$

For closer memory structures, their runtime estimate is the same as produced by a worst case analysis and applying equations 6.1 and 6.2. The result from the worst case analysis is compared to the best case prediction and the greater of the two is used as the runtime estimate.

6.3.2 Proof of Best and Worst Case

For the best case, we assume that data are always moved from the memory structure with the highest data movement cost to the processor whenever possible. We divide the proof into two cases, structures larger than the bound memory structure and those smaller than the bound structure. For structures larger than the bound structure, reads are included in the reads to the bound structure. Therefore, because the cost of moving data from the larger structure is less than the bound structure, the data from the larger structure can be moved to the bound structure while the bound structure is moving data to the processor and does not impact runtime.

For structures smaller than the structure with the largest data movement cost, data that are not read from the bound structure can be moved to the processor when there is spare bandwidth. The reads can be interleaved between reads from the bound structure or travel on a different bus from the smaller structure to a different processing core. However, certain data access patterns prevent data movement from a closer cache to a processor from being performed in a manner that does not impact performance. When the data movement cost can not be hidden we can not use the best case estimate.

Instead, when the runtime of the bound structure times the number of cores that share a bus from the bound structure is less than the worst case analysis, we use the worst case analysis. When this inequality is not true, the calculation is executed in a way such that it is impossible to overlap data from closer caches.

In the worst case, data movement is assumed to cause conflicts for resources whenever pos-

sible. We prove that having all cores execute the same code simultaneously is the worst possible execution pattern by contradiction. We assume that multiple cores are executing a different part of the program at once and that this execution pattern results in the most possible conflicts. We also assume each core needs to move data over the same memory buses and that one memory bus is used at full capacity. The bus limiting data movement can be described using two cases.

In the first case, the combined execution of the codes causes the bus closer to the processor to be used at full capacity. One of two scenarios or a combination of both can occur. In the first scenario, codes not bound on data movement from the bus run at the same speed as if each core were executing them. The core bound on data movement over the bus at capacity uses the excess bandwidth not used by the other cores to execute calculations faster than if all cores were executing the same code. In the second scenario, cores bound on data movement over the bus at capacity run at the same speed as they would if all cores were executing the same code. The cores that would be bound on a bus farther from the processor if all cores were executing the same code as each other can execute faster. They can use the excess bandwidth on the bus farther from the processor. In both scenarios one or more cores executes more code than if both cores executed the same code creating a contradiction.

In the second case, the combined calculation is bound on data traveling over a bus farther from the processor than part of the calculation which is bound on another bus closer to the processor. In this case there are once again two scenarios. In the first scenario, codes not bound on data movement from the bus run at the same speed as if each core was executing them. Since these cores do not use as much of the bandwidth from the farther memory structure, there is more available to execute the calculation bound on data movement over the farther memory structure. In the second scenario, cores bound on data movement over the bus at capacity run at the same speed as they would if all cores were executing the same code. The cores running calculations not bound on data movement from the farther structure are able to use the extra bandwidth available between closer memory structures and the processor to run faster than if all cores were executing the same calculation. In both scenarios, in this case more computation occurs if cores execute

different calculations than the same calculation at once. Therefore, having all cores execute the same code at once produces the worst possible data movement patterns.

6.3.3 Implementation

For converting memory traffic estimates to runtime predictions on parallel machines, both the best and worst case execution patterns must be modeled. We solve this problem by creating two runtime prediction methods. The worst case prediction method performs the first execution pattern presented in Section 6.1.2, and the best case method uses the second execution pattern. Both of these prediction methods account for the same hardware features but assume different calculation execution paths. Hardware features modeled include the amount of bandwidth available between each memory structure and core, how many data are moved over that bandwidth and how many cores share that bandwidth.

The worst case prediction method uses a modified version of the single processor runtime prediction method described in Section 5.2.2, accounting for the increase in bandwidth created by multiple buses. The serial algorithm is used to traverse the AST. In parallel the serial traversal scheme implies that all cores are executing the same loop of a calculation at the same time. Simultanious execution is implied because all runtime predictions are calculated for each loop . To account for parallel structures, bandwidths from each cache to processing cores are multiplied by the number of buses from a memory structure to cores. For the example Clovertown machine in Figure 6.1, the bandwidth between memory and the cores is multiplied by two and the bandwidth between the L1 and L2 caches is multiplied by eight. The total number of misses for all cores to a memory structure for all inner loops is divided by the bandwidth to produce runtime estimates for all inner loops. These runtime estimates are propagated up the AST in the same manner as in the serial algorithm to produce overall runtime predictions.

The best case prediction method assumes that data are always read from the memory structure that limits performance. This assumption implies that computations that can be overlapped always are. The predicted runtime is found by summing all misses to each memory structure. The

sum of the number of misses to each memory structure is then divided by the bandwidth between that memory structure and the processor in the cost algorithm. The maximum of these values is the runtime estimate. There is one exception in the algorithm as explained in the previous section. When only a single core accesses a memory structure over a given bus and another memory structure is between it and the CPU, the worse case prediction method is used for those memory structures since calculations cannot be overlapped. That cost is then compared to the cost of moving data through memory structures where overlapping can occur, with the greater of the two selected.

In practice, when executing a calculation where overlapping is possible, the execution results in some contention and some overlapping. Therefore, we use our best and worst case estimates in combination to provide high and low bounds on potential performance.

6.3.4 Results

The second step in evaluating the accuracy of our parallel memory model was comparing runtime predictions to the actual routine runtimes. We ran tests using parallel versions of routines produced by BTO on the Work and Opteron machines. In this section we present selected results that are typical of performance on these two machines.

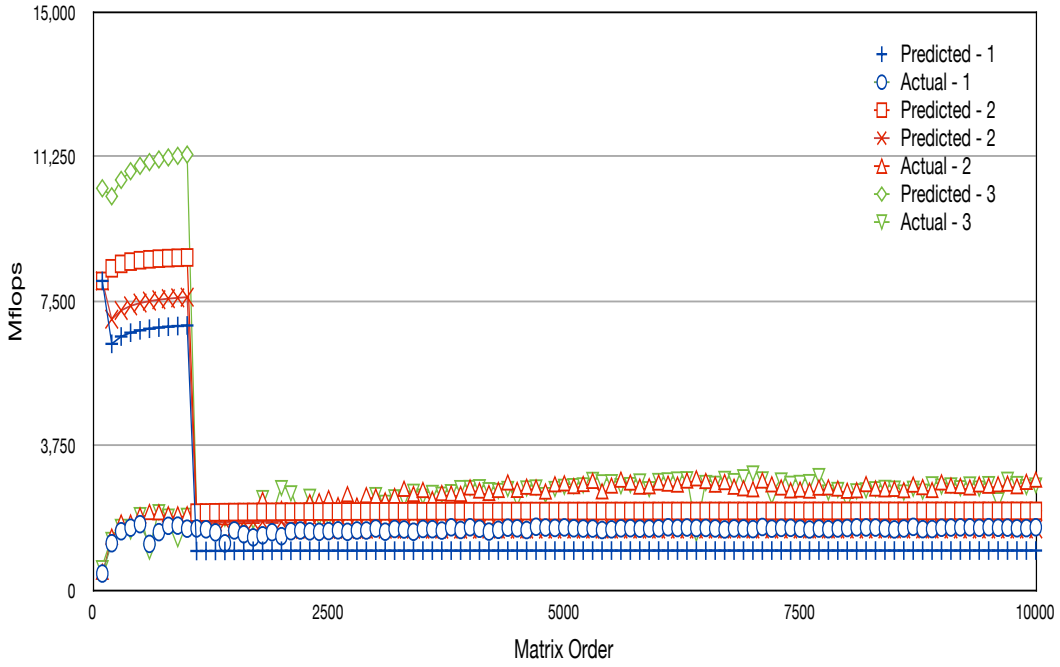
Figures 6.4 and 6.5 show runtime estimates and measured performance on the Work machine. In these figures and the rest in this section routines are numbered with one representing the least amount of fusion and routines with progressivly higher numbers containing more fusion. On the Work machine for matrix kernels Figure 6.4 displays that our model's runtime predictions were not as accurate at predicting actual performance as they were in serial. However, the runtime predictions are useful in determining performance differences of different versions of the same routine. Being able to accurately distinguish performance differences between versions based on their memory costs means that our model can be used to determine which versions produced by BTO will perform the best. Figure 6.5 shows that on the Work machine our model for large vector dimensions accurattly predicts routine runtime. Predictions for all versions are within 10% of their

actual values for all the versions shown.

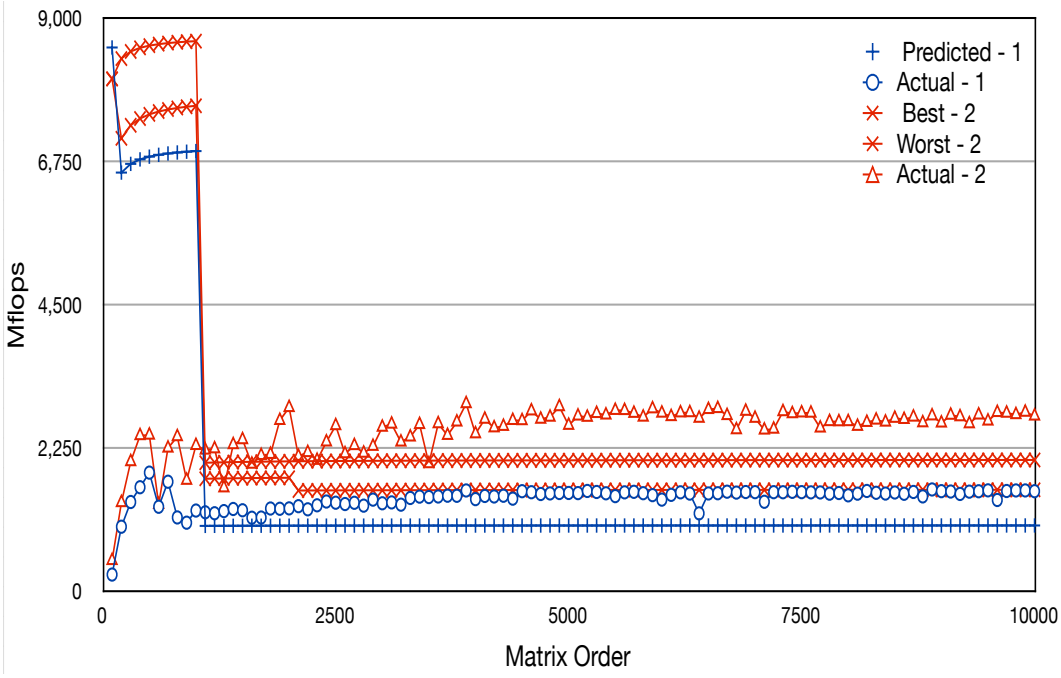
Figures 6.6 and 6.7 show runtime estimates and measured performance on the Opteron machine. Predictions on the Opteron are within 10% for large matrix orders and vector dimensions. The actual values have more variability than on the Work machine. The variability is probably due to the NUMA archnitecture with threads changing which processor they are exectuting on during execution. Pinning threads to a single processor might reduce the variability of runtimes, but for now BTO does not support pinning.

On both machines for matrix based kernels the speed of the actual kernels increases with matrix order. Our model does not capture this effect because it only profiles the hardware and mea sures usable bandwidth at one size. To capture the increase in speed in our runtime predictions we would need to profile the machine at various sizes and interpolate those values.Using this more accurate benchmarking procedure for various kernel sizes might improve our predictions, but the benefits need to be weig hed against the added costs of measuring more values at install time and looking up values at runtime. However, as implimented the relative accuracy of our predictions demonstrates that the model can be used to accurately and efficiently trim the search space of versions of routines considered within BTO as shown in the next chapter.

Also on both machines actual performance for small kernels is significantly less than predicted. There are two reasons for the gap. As with our serial predictions the parallel model does not account for the cost of arithmetic. The new factor not accounted for in parallel is the amount of time it takes to generate threads for parallel computation.

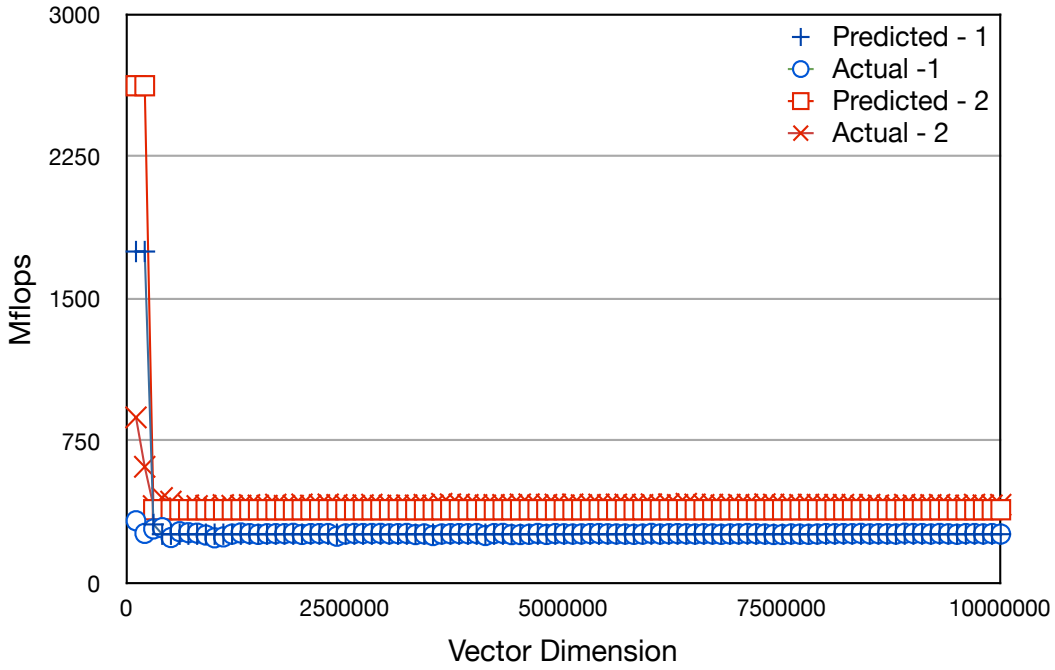


(a) BiCGK

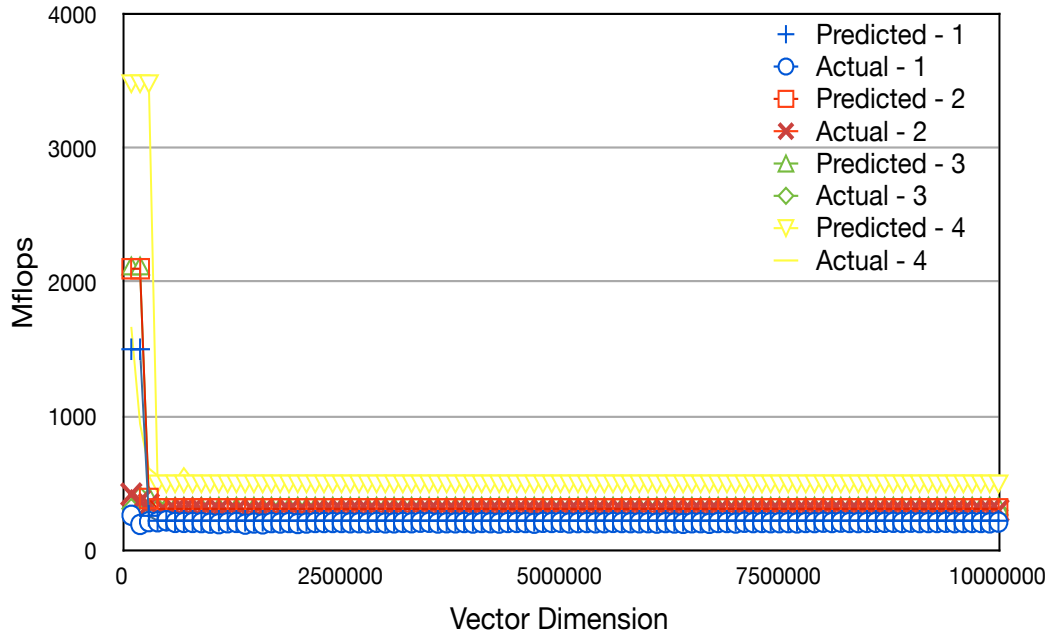


(b) AATX

Figure 6.4: Actual and predicted runtimes comparison for matrix kernels on the Work machine.

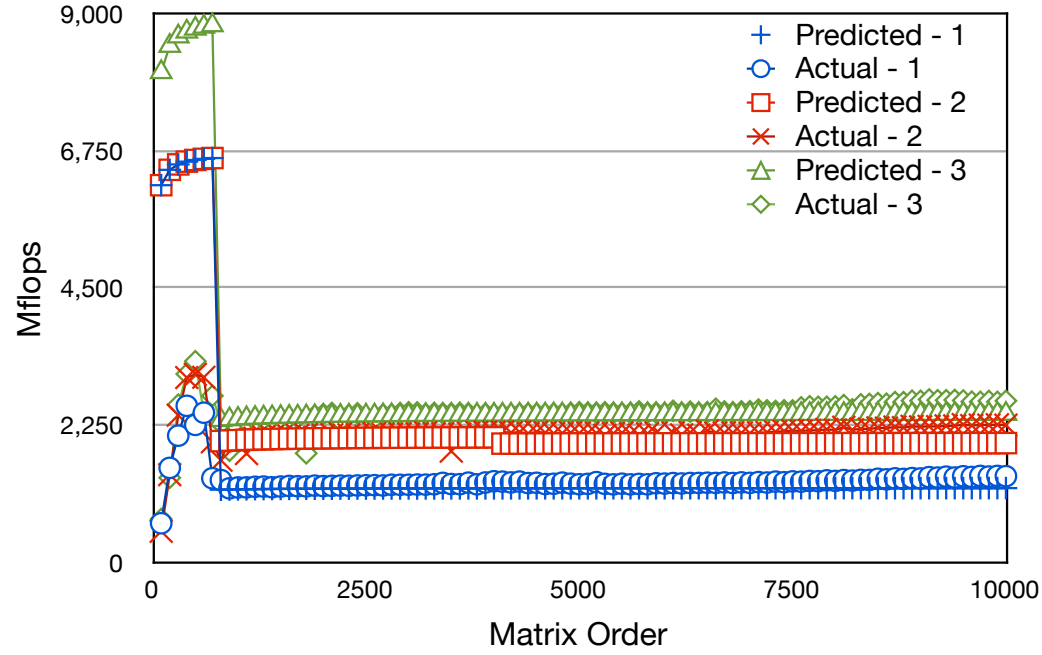


(a) VADD

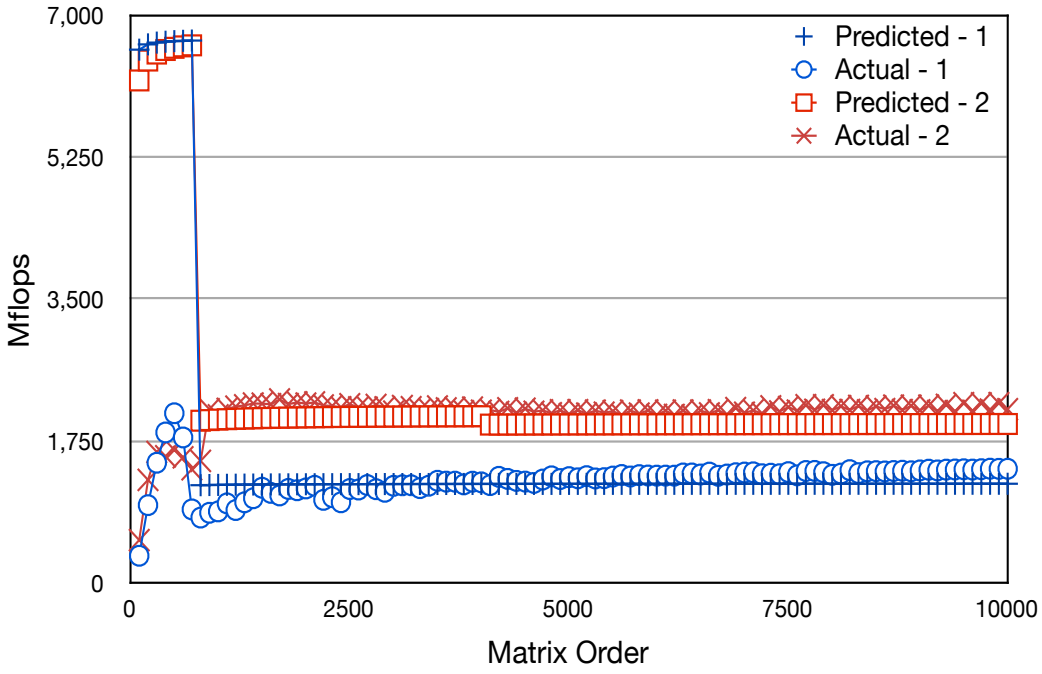


(b) WAXPBY

Figure 6.5: Actual and predicted runtimes comparison for vector kernels on the Work machine.

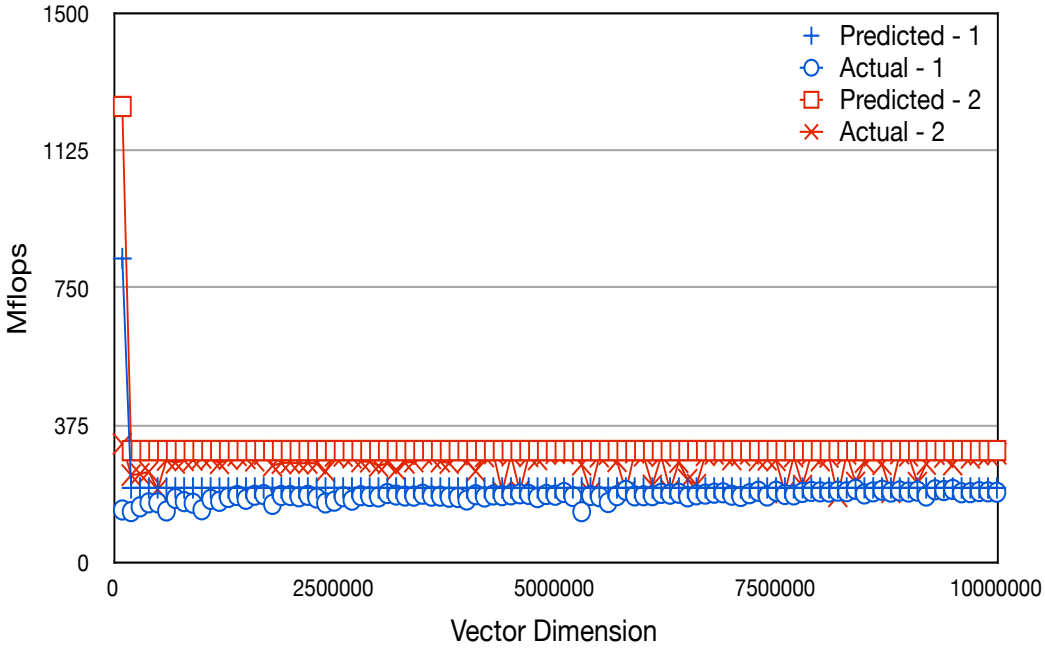


(a) BiCGK

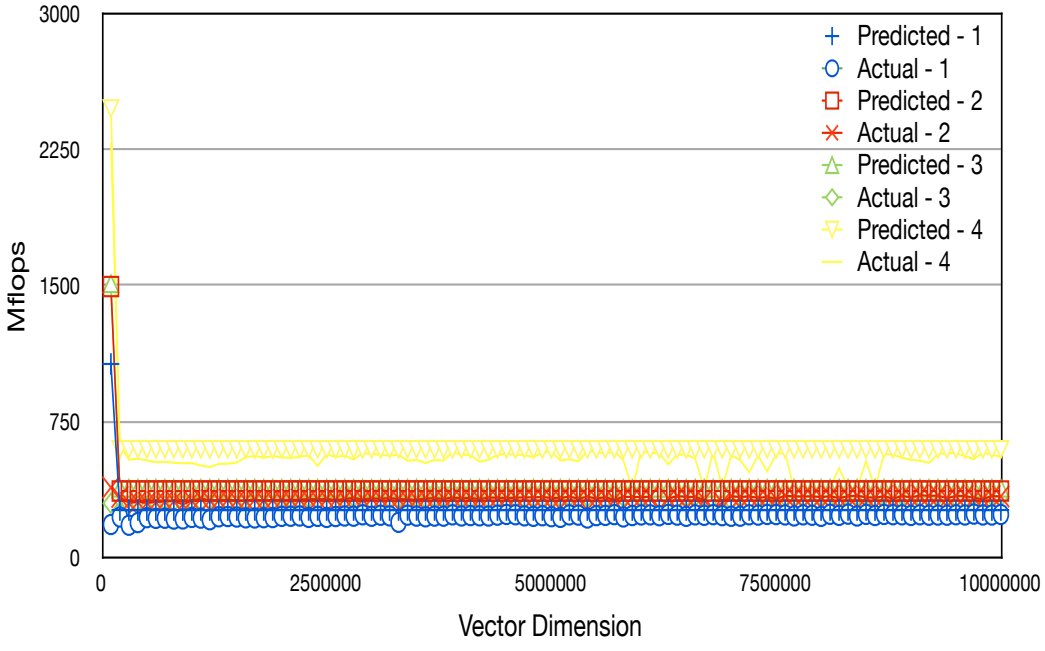


(b) AATX

Figure 6.6: Actual and predicted runtimes comparison for matrix kernels on the Opteron machine.



(a) VADD



(b) WAXPBY

Figure 6.7: Actual and predicted runtimes comparison for vector kernels on the Opteron machine.

Chapter 7

Integration of Runtime Prediction into BTO Compiler

The serial and parallel memory models described in this thesis are integrated into the BTO compiler within which they reduce the amount of time used to empirically test different versions of the same routine. In this chapter, we describe how the models are integrated into BTO and how we use them to reduce search time in BTO. Throught the chapter we compare and contrast search time and the produced routine's performance using the model in conjunction with empirical search to exhaustive empirical search.

7.1 Integration of the Model into BTO

To include the models in BTO, we first convert the internal graph representation of a routine, used by the compiler, to the tree format used by the model. Different representations are used because BTO requires information to make decisions that the model does not need. The less complex tree representation used by the model adds convenience and speed. Next, the compiler calls a `create_machine` function, which generates a machine structure for the current system and model found at install time, as described in Section 5.1. Then the compiler invokes an interface function that runs the memory traffic prediction and cost functions. The interface function returns runtime estimates for the input routine and machine pair. In serial, a single estimate is returned, and, in parallel, the best and worst case predictions are returned. These values are then placed in a vector structure sorted by model predictions.

The BTO compiler has five flags that control how the model and empirical testing interact

when searching for high performing routines. Two of these flags `-m [ --model_off]` and `-e [ --empirical_off]` turn off one evaluation method. The other three control the routines and the sizes that are empirically tested. The flag `-l [ --limit]` specifies a maximum amount of time in seconds to be used for empirical search. When selected, the empirical search begins with the routine predicted best by the model and continues until the time limit is reached. The size(s) of routines to model and empirically test are set by `-r [ --test_param]`. How aggressively the model trims the search space is determined by the `-t [ --threshold]` flag. This flag restricts the compiler to empirically test only those versions within a set percentage of the version predicted best by the model.

The use of these flags can be both advantageous and detrimental to compiler runtime and produced routine performance. To decrease compile time, empirical testing can be turned off producing the version of the routine the model predicts as best. While substantially reducing runtime, not empirically testing routines can result in subpar performance of the produced routine, as shown later in this chapter. A second way to reduce compile time is to order all versions based on the model's predictions and then test them in that order until a given amount of time elapses. By specifying a maximum amount of time for empirical testing, compilation is guaranteed to finish in an acceptable amount of time. However, routines with poor predicted and actual performance might be tested thus wasting time.

Another way to trim search time is only empirically testing routines predicted by the model to have runtimes within a certain percentage of the best predicted runtime for a version. Only emperically testing the best predicted versions impacts compile time and produced routine performance to varying degrees, depending on the threshold set. The threshold and time limit flags can be used together with the advantages and disadvantages of each being combined. If time is not a concern, then all routines can be empirically tested by turning off the model or setting the threshold parameter to one. Finally, if a user knows that, on their machine, routines of varying sized kernels perform similarly, empirical tests can be performed on smaller test sizes than the target kernel size. Empirical testing time is reduced and/or more kernels can be tested in the same amount of time,

Kernel	Serial Versions	Parallel Versions
AATX	2	4
BiCGK	3	8
DGEMV	4	18
DGEMVT	8	31
GEMVER	648	7808
GESUMMV	12	52
HOUSE	6	24
AXPYDOT	4	9
GRAMM	2	3
VADD	2	3
WAXPBY	4	9

Table 7.1: The number of serial and parallel versions produced by the BTO compiler for various routines.

but the user risks a suboptimal kernel being produced.

7.2 Analysis of Model's Effectiveness in Reducing Compile Time

The model and runtime prediction function are only useful to the BTO compiler if they reduce overall search time without compromising the quality of the produced routines. In this section, we analyze the model's effectiveness at reducing the search space and compare the model's speed to empirical testing. We include discussion of the model's strengths and weaknesses along with performance results in our evaluation.

7.2.1 Experimental Setup

We tested the model's ability to trim the compiler's search space for the latest release of the BTO compiler (Version 1.2) on the kernels listed in Table 3.2. The selected kernels were chosen for their diversity of routine features, real world use and varying search space sizes. Tests were performed on the Work and Opteron systems in Tables 3.1 and 6.1 for both serial and parallel routines. The number of versions produced by BTO for each of these routines is shown in Table 7.1. The kernels at the top of the table contain matrices and vectors and the kernels at the bottom include only vectors. All the tables in this chapter are divided in this manner.

We ran all the kernels on both machines and measured the amount of time used by the model to predict the runtimes of all versions of a routine. We also measured the amount of time needed to empirically test all produced versions.

7.2.2 Cost of Modeling and Empirically Testing Runtimes

Tables 7.2 and 7.3 show the number of versions of a routine that our model and empirical testing analyze per second on the Work and Opteron machines. The table shows that the model predicts performance of routines hundreds to thousands of times faster than it takes to empirically test them. Also shown in the tables is that our model's runtime varies little based on data structure size, but the amount of time needed to empirically test routines is proportional to the dimensions of the matrices and vectors tested.

From the tables, we observe that both empirical testing and the model are slower for more complex routines. For example, GEMVER, which is the most complex routine containing matrices is the slowest to model and empirically test for both machines in parallel and serial. The parallel model runs at least 25% slower than the serial model. The parallel model is slower because it calls a second runtime prediction function and parallel machines are more complex. Empirical testing of routines occurs at approximately the same speed whether the routine is serial or parallel. Finally, not shown in the table but of note is that the model evaluates each version of a routine three to four times faster than the compiler generates them on both machines.

Figure 7.1 shows that the runtime of the model is fairly independent of matrix order. When GEMVER is modeled for matrix orders ranging from one to one million on the Work machine, there are only slight runtime differences as size changes. For both the serial and parallel model, runtimes increase with routine size, by only 20% to 30%.

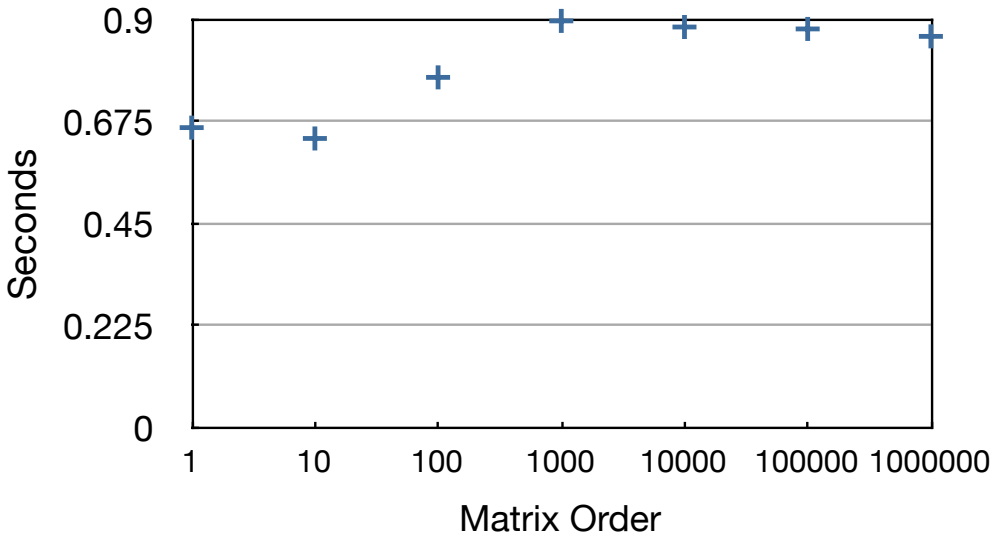
Both models follow the same pattern with runtime increasing as routine size increases up to one thousand and then slightly decreasing as routines get larger. The increase for small matrix orders occurs as the calculation becomes larger than successive caches, increasing the number of caches for which misses must be calculated. From a matrix order of ten to one hundred, the matrices

Kernel	**Dimension**	**Model Serial/Parallel**	**Empirical Serial/Parallel**
AATX	2000	4843/3344	2.37/2.19
	10,000	4890/3309	0.20/0.20
BICG	2000	5396/3501	2.40/2.17
	10,000	5396/3492	0.19/0.20
DGEMV	2000	3252/2575	2.35/1.79
	10,000	3147/2497	0.20/0.17
DGEMVT	2000	2092/1406	2.28/2.03
	10,000	2116/1413	0.19/0.19
GEMVER	2000	734/456	1.31/1.14
	6000	733/455	0.23/0.23
GESUMMV	2000	2210/1431	1.59/1.47
	10,000	2224/1431	0.10/0.11
HOUSE	2000	2326/1622	2.21/1.99
	10,000	2325/1614	0.17/0.18
AXPYDOT	2,000,000	5487/3314	1.70/1.57
	10,000,000	5479/3308	0.47/0.46
GRAMM	2,000,000	5076/3128	2.43/2.23
	10,000,000	5115/3128	0.81/0.79
VADD	2,000,000	7905/5848	1.63/1.65
	10,000,000	7782/5747	0.46/0.47
WAXPBY	2,000,000	5706/3965	1.96/1.77
	10,000,000	5755/3956	0.56/0.59

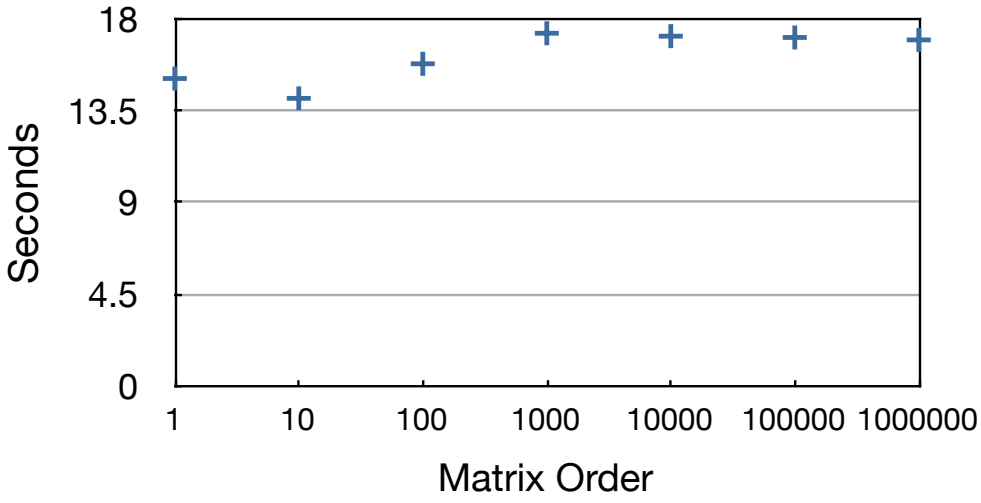
Table 7.2: Number of routines the model and empirical testing evaluate per second on the Work machine.

Kernel	**Dimension**	**Model Serial/Parallel**	**Empirical Serial/Parallel**
AATX	1000	2058/1500	1.57/3.58
	10,000	2058/1466	0.16/0.18
BICG	1000	2445/1574	1.71/3.60
	10,000	2368/1587	0.17/0.19
DGEMV	1000	1494/824	1.55/2.49
	10,000	1440/830	0.18/0.16
DGEMVT	1000	891/610	2.35/2.81
	10,000	943/607	0.16/0.19
GEMVER	1000	763/458	1.07/1.80
	10,000	752/466	0.19/0.22
GESUMMV	1000	986/620	2.65/2.35
	10,000	966/620	0.10/0.10
HOUSE	1000	1035/697	2.31/2.70
	10,000	1027/687	0.15/0.17
AXPYDOT	1,000,000	2509/1483	1.80/1.65
	10,000,000	2536/1483	0.38/0.41
GRAMM	1,000,000	2291/1372	1.60/1.96
	10,000,000	2210/1386	0.63/0.65
VADD	1,000,000	3305/2573	1.38/2.31
	10,000,000	3521/2560	0.41/0.37
WAXPBY	1,000,000	2559/1824	2.11/2.10
	10,000,000	2492/1832	0.45/0.50

Table 7.3: Number of routines the model and empirical testing evaluate per second on the Opteron machine.



(a) Serial



(b) Parallel

Figure 7.1: Time to model all versions of GEMVER on Work machine.

become larger than the L1 cache and from one hundred to one thousand they become larger than the L2 cache. For a matrix order of one, the model takes longer to run than for an order of ten. While modeling for such a small size is not practical, the extra runtime is probably attributable to modeling register misses for more variables. Within the model the code to model register misses is more complex than the code to model cache misses, which most likely increases the runtime. Finally, the extra predicted runtime produced by the parallel model reduces variance by adding a fixed cost to the parallel model.

7.2.3 Model's Impact on Serial Compile Time

Tables 7.4 and 7.5 show the impact of the serial model on routines tested and compiler time on the Work and Opteron machines. The runtime column reports the amount of time it takes to generate the routines, perform model predictions for all versions of the routine and to empirically test the best predicted versions. The savings column reports the compile time reduction from using the model in hybrid search, as opposed to empirically testing all routines. Positive values represent savings and negative values represent increased costs. For both tables, different versions of routines are only empirically tested if their predicted runtime is within 1% of that of the best predicted routine. When a single routine is predicted to be the best routine, it is not tested.

In all but one case, the optimal routine was either the single routine predicted best by the model or in the group of best predicted routines. Compile times are also reduced by over 99% for all but a few routines when using hybrid search. For the routines where multiple versions were empirically tested, compile time was still reduced by half for AXPYDOT and slightly increased for DGEMV and GESUMMV.

For a matrix order of 6000 on the Work machine, the GEMVER routine predicted best by the model was 2% slower than the optimal routine found by empirical search. The optimal routine was predicted to be just over 8% slower than the best predicted routine and was found in a group of seven routines predicted to have nearly identical performance. This cluster of versions was ranked as the second best through eighth best by the model and empirically testing the first eight routines

Kernel	**Size**	**Empirically Tested**	**Runtime**	**Model Savings**
AATX	2000	0	0.020413	0.822587
	10,000	0	0.020409	10.224591
BiCGK	2000	0	0.021556	1.249444
	10,000	0	0.021556	15.756444
DGEMV	2000	4	1.723230	-0.01230
	10,000	4	19.941271	-0.01271
DGEMVT	2000	0	0.033825	3.501175
	10,000	0	0.033781	41.130219
GEMVER	2000	0	3.528834	492.419
	6,000	0	3.529889	2877.409
GESUMMV	2000	12	7.587429	-0.005429
	10,000	12	123.887396	-0.005396
HOUSE	2000	0	0.029580	2.706420
	10,000	0	0.029581	34.920419
AXPYDOT	2,000,000	2	1.175	1.200
	10,000,000	2	4.236	4.360
GRAMM	2,000,000	0	0.020394	0.822606
	10,000,000	0	0.030391	2.481609
VADD	2,000,000	0	0.020253	1.227747
	10,000,000	0	0.020257	4.325743
WAXPBY	2,000,000	0	0.022701	2.042299
	10,000,000	0	0.022695	7.149305

Table 7.4: Impact of model on reducing search space and runtime for serial routines on the Work machine.

Kernel	**Size**	**Empirically Tested**	**Runtime**	**Model Savings**
AATX	1000	0	0.044972	1.27003
	10,000	0	0.044908	12.4001
BiCGK	1000	0	0.047227	1.75277
	10,000	0	0.047267	17.5447
DGEMV	1000	4	2.63568	-0.008974
	10,000	4	22.8628	-0.008484
DGEMVT	1000	0	0.064974	3.39703
	10,000	0	0.064484	49.1745
GEMVER	1000	0	3.29977	606.778
	6,000	0	0.861650	3429.02
GESUMMV	1000	12	4.59417	-0.012167
	10,000	12	125.288	-0.012423
HOUSE	1000	0	0.056798	2.58520
	10,000	0	0.05684	40.7542
AXPYDOT	1,000,000	2	1.498	0.719
	10,000,000	2	5.272	5.232
GRAMM	1,000,000	0	0.044873	1.24613
	10,000,000	0	0.044905	3.18510
VADD	1,000,000	0	0.043605	1.44940
	10,000,000	0	0.043568	4.88432
WAXPBY	1,000,000	0	0.045563	1.89744
	10,000,000	0	0.045695	8.90040

Table 7.5: Impact of model on reducing search space and runtime for serial routines on the Opteron machine.

would increase compile time to 43.169 seconds.

On both machines, there are three routines - DGEMV, GESUMMV and AXPYDOT - where the model groups multiple versions of the same routine together. For DGEMV, the grouping is accurate because on both machines and for all matrix orders, the performance difference of the kernels produced was less than 3%. However, for AXPYDOT, performance differences between the two best predicted versions were 15% on the Opteron and 20% on the Work machine. For GESUMMV, performance differences were less than 3% on the Work machine for all versions tested, but on the Opteron the gap between the best and worst routines is over 50% for a matrix order of 1000 and 30% for a matrix order of 10,000. On the Opteron, the GESUMMV versions clustered into three groups of four routines with near identical performance. Within the groups the routines only differ by the fusion of a vector operation. Therefore, if we applied our result in section 3.6 to GESUMMV, we would test three routines and not twelve to find one of the best performing routines. The same research can also be applied to DGEMV, reducing its search space to a single routine on both machines.

The model was not ideal in two cases. It missed the best routine, using our 1% criteria, for one matrix order of GEMVER on the Work machine and failed to differentiate between versions of GESUMMV on the Opteron with 30-50% performance differences. For GEMVER, depending on how much a user was planning on using the produced kernel and the amount of time they had for compilation, our results might be satisfactory. Alternatively, a small increase in compile time would quickly find the best routine. Not reducing the search space of the GESUMMV routine highlights a weakness of our approach. We only try to capture large memory differences caused by loop fusion and can miss smaller effects.

7.2.4 Model's Impact on Parallel Compile Time

Tables 7.6 and 7.7 show the impact the parallel model has on reducing the number of routines tested and compile time on the Work and Opteron machines. In the tables, the columns contain the same information presented for the serial case with one exception. In some cells we present two

values for the number of routines tested, runtime and model savings since we use two criteria to determine which routines to empirically test. The first criterion is the average of the best and worst case estimates and the second is the best case estimate. As in our serial experiments, we assume different versions of routines are empirically tested only if their predicted runtimes are within 1% of the best predicted routine.

On the machines in parallel, compile time was greatly reduced with savings on both machines exceeding 99% for many routines. Additionally, in our parallel tests, the optimal routine was the best predicted version or found in a group of the best predicted with three exceptions. On the Work machine, the optimal GEMVER implementation for a matrix order of 2000 was 1% faster than the best kernel in the model's best predicted group. The optimal routine was predicted to be just under 17% slower than the best predicted routine and was found in a group of 61 routines predicted to have nearly identical performance. This cluster of versions was ranked as the 32nd best through 92nd best by the model. Testing the first ninety-two routines would increase compile time by 64.922 seconds. On the Opteron with a matrix order of 2000, the best routine was not found for the GESUMMV and DGEMV routines. The routine found was 23% slower than the optimal for GESUMMV and 11% slower for DGEMV. For both routines, increasing the threshold to 2% would have tested all versions produced by the compiler and found the optimal implementation. The number of additional routines tested would increase by four for DGEMV and sixteen for GESUMMV.

For the routines on the Work machine where empirical testing was used along with modeling, the performance of the tested versions varied more than in serial. The versions of DGEMV and GESUMMV tested saw performance variations of about 25% for a matrix order of 2000. For a matrix order of 10,000, DGEMV did not have large performance differences between versions, but GESUMMV had four versions perform about 25% slower than the rest. AXPYDOT had about a 15% performance difference between the two kernels tested for both vector dimensions tested. For BiCGK, the best kernel is found in the group of the two kernels with the best predicted runtime when best and worst cases predictions were averaged for a matrix order of 10,000. Within this

Kernel	**Size**	**Empirically Tested**	**Runtime**	**Model Savings**
AATX	2000	0	0.028196	1.82280
	10,000	0	0.028209	20.1668
BiCGK	2000	2/4	0.995485/1.88149	2.72252/1.83652
	10,000	2/4	9.99729/19.9873	30.2127/20.2227
DGEMV	2000	22	10.1360	-0.006991
	10,000	22	109.125	-0.007210
DGEMVT	2000	0	0.123053	15.2779
	10,000	0	0.122946	159.156
GEMVER	2000	30/31	111.387/111.872	6738.81/6738.32
	6,000	0	206.047/209.456	34,393.9/34,390.5
GESUMMV	2000	52	35.5043	-0.036337
	10,000	52	493.943	-0.036328
HOUSE	2000	0	0.092797	12.0692
	10,000	0	0.092788	136.786
AXPYDOT	2,000,000	2	1.25127	4.47573
	10,000,000	2	4.261271	15.3687
GRAMM	2,000,000	0	0.025959	1.34204
	10,000,000	0	0.025977	3.80302
VADD	2,000,000	0	0.023513	1.81749
	10,000,000	0	0.023513	6.39778
WAXPBY	2,000,000	0	0.034270	5.08973
	10,000,000	0	0.034275	15.1957

Table 7.6: Impact of model on reducing search space and runtime for parallel routines on the Work machine.

Kernel	**Size**	**Empirically Tested**	**Runtime**	**Model Savings**
AATX	1000	0	0.058666	1.11433
	10,000	0	0.058728	22.0533
BICGK	1000	2	1.149	1.131
	10,000	2	10.737	31.122
DGEMV	1000	18	6.795	0.421
	10,000	22	114.063	-0.021688
DGEMVT	1000	0	0.203885	10.9771
	10,000	0	0.204357	164.424
GEMVER	1000	30	89.439	4307.01
	6000	30	198.612	35293
GESUMMV	1000	36	15.604	6.553
	10,000	52	508.904	-0.083913
HOUSE	1000	0	0.161424	8.84058
	10,000	0	0.161914	141.207
AXPYDOT	1,000,000	3	2.170	3.355
	10,000,000	3	7.110	14.727
GRAMM	1,000,000	0	0.054186	1.53081
	10,000,000	0	0.054164	4.57884
VADD	1,000,000	0	0.048166	1.29683
	10,000,000	0	0.048172	8.01283
WAXPBY	1,000,000	0	0.062934	4.28307
	10,000,000	0	0.062913	18.0741

Table 7.7: Impact of model on reducing search space and runtime for parallel routines on the Opteron machine.

group, runtimes vary by 10%. However, for a matrix order of 2000, the best kernel is not found until all four routines with similar best case runtimes are included in the empirical testing. Within this group of four kernels, performance varies by 20%. For the GEMVER kernel, both criteria produced the same routine for both matrix orders. Runtimes of the routines clustered into three groups for a matrix order of 2000. Of note is that all kernels with the best predicted performance were in the first group.

On the Opteron, both the average and best predicted time criteria produce groups of equal size. For AXPYDOT, in the best predicted group, there is a 17% performance difference between the best and worst kernel for the 1,000,000 test size and a 9% difference for the 10,000,000 size. Performance for both the DEGEMV and GEMVER kernels varies greatly for a matrix order of 1000 with the slowest routine taking more over 150% the time as the fastest. However, for a matrix order of 10,000, DGEMV sees performance differences of less than 3% and, for a matrix order of 6000, GEMVER performance varies by less than 10%. For GESUMMV, routines with a matrix order of 10,000 actual routine runtimes are in two clusters separated by about 15%, while for a matrix order of 1000 performance varies by just over a factor of two from best to worst. BICGK has a small runtime difference of less than 3% for a 10,000 matrix order and a difference of just under 10% for a matrix order of 1000.

As with our serial model, the parallel model failed to distinguish between versions with significant performance differences for the GESUMMV and DGEMV kernels on the Opteron. The parallel model also produced the second best routine on the Work machine for one matrix order of GEMVER. However, as in the serial case, the model dramatically reduced compile time for most routines with only a minimal reduction in the quality of the produced kernel.

Chapter 8

Conclusions

Data movement through the memory hierarchy often limits linear algebra routine performance. For these routines, reducing data traffic often results in significant speedups. Throughout this thesis, our focus is on loop fusion, which is one optimization used to minimize data reads and writes from slow memory. We show the positive and negative impacts of fusion on data movement and routine performance for linear algebra kernels. How loop fusion affects reads and writes from memory structures is the basis for the memory model we present.

Our model works in two steps. First, it predicts the amount of data movement from each memory structure needed to execute a linear algebra routine. Then the model converts those estimates into runtime predictions in seconds. These runtime predictions are used on both serial and parallel machines to compare different implementations of the same calculation within the BTO compiler. When turned on, the model usually reduces compile time by over 99%, while having negligible impact on the quality of the routine BTO produces.

The model achieves these compile time reductions at a small accuracy cost by using a series of tradeoffs between speed and accuracy. For example, we incorporate cache size, data transfer rates and how many processors share a given cache into the model but do not include cache associativity, latency of data reads and the cost of arithmetic instructions. These tradeoffs result in a model that is efficient in reducing the number of routines empirically tested by our compiler but the model has a few weaknesses. For almost all the kernels used to test our model, compile time was significantly reduced and a high performing routine produced. However, this was not the case

for the DGEMV and GESUMMV calculations. When vector operations are fused with matrix operations, routine performance is usually not impacted. However, since the model is ineffective at differentiating between routines with small performance differences, we empirically test these routines. Also, on the Opteron machine, there was a significant performance difference between versions of GESUMMV but the model predicted them to perform equally. Another way to reduce the compile time of DGEMV and GESUMMV is to remove fusion decisions that never positively impact routine performance from the search space.

Overall, the model achieves the task it is designed to handle. It accurately and efficiently distinguishes between large routine performance difference for most routines and dramatically reduces the runtime of the BTO compiler.

Chapter 9

Future Work

One area where our model can be improved includes increasing the variety of machines for which it predicts the runtimes of loop fusion kernels. For example, fusion when applied to OpenCL [50] kernels on graphical processing units (GPUs) is effective at reducing data movement [12]. As with CPUs, the performance of GPUs is often limited by reads and writes. Additionally, as with the CPU, fusing many kernels on the GPU can decrease performance. To model GPUs requires the ability to analyze data movement within the GPU and data transfers between a computer's main memory and the GPU. Also needed is a method to determine, at install time, the amount of useable bandwidth between the CPU and GPU and within the GPU.

Another new machine class that can be modeled is clusters and other distributed memory machines [75]. Estimating runtime on clusters requires adding latency, which is often a larger cost of data movement than bandwidth between distributed nodes, to the model. Also, a way to indicate segments of code that can execute while data transfers occur is needed because overlapping data movement and computation is common in distributed memory programs.

Other enhancements to the model, with applicability to many computing environments, include the ability to analyze strided data access patterns and recommend cache block sizes. Strided access patterns occur in many mathematical computations, such as transposing a matrix, and recently the BTO compiler was improved to allow the production of codes with strided accesses [9]. To predict the cost of strided accesses, cache line sizes and/or data movement rates for various strides are needed and should be found at install time. Strided access patterns create additional

data movement because they only use part of a cache line, but they require the entire line to be moved through the memory hierarchy. To model strided patterns, an additional field denoting variables that are accessed using a strided pattern needs to be added to variable representations. Also modifications to the model are required to count the extra data moved but not used when only part of a cache line is used.

The model is currently able to predict the memory traffic of code with cache blocks, but it cannot determine if the proper size was chosen. Adding the ability to determine the maximum size of blocks that allows data to fit within cache might improve search times for optimal block size depending on the additional costs.

To improve model runtime, different versions of the same routine can be evaluated in parallel. For parallel evaluation to significantly impact compile time, BTO needs to generate routines in parallel because routine generation is a larger cost than modeling. However, searching for cache block sizes in parallel with the current model could benefit from parallel evaluation by dedicating a thread to evaluating different block sizes for the same version of a routine.

Finally, search strategies other than exhaustive search, such as genetic algorithms, are being evaluated for their use within BTO [77]. When using the model in these alternative search strategies, different tradeoffs between runtime and accuracy might be beneficial. The tradeoffs used would vary depending on their strengths and weaknesses and how the search strategies use the model.

Bibliography

[1] Mark F. Adams. Multigrid Equation Solvers for Large Scale Nonlinear Finite Element Simulations. PhD thesis, University of California, Berkeley, Berkeley, CA, January 1999.

[2] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. ACM Computing Surveys, 27(3):367–432, September 1995.

[3] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh cfd application. In Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), Supercomputing '99, Portland, Oregon, United States, 1999. ACM.

[4] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. DuCroz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorenson. LAPACK: A portable linear algebra library for high performance computers. In Proceedings of Supercomputing '90, pages 2–11, New York, NY, November 1990.

[5] A. H. Baker, J. M. Dennis, and E. R. Jessup. An efficient block variant of GMRES. SIAM J. Sci. Comput., 27:1608–1626, 2006.

[6] A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted GMRES. SIAM J. Matrix Anal. Appl., 26:962–984, 2005.

[7] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc users manual. Technical Report ANL-95/11–Revision 2.1.2, Argonne National Laboratory, Argonne, IL, 2002.

[8] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. Templates for the solution of linear systems: Building Blocks for Iterative Methods. SIAM, second edition, 1994.

[9] Geoffrey Belter. Personal communication, December 2010.

[10] Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. Automating the generation of composed linear algebra kernels. In SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pages 1–12, New York, NY, USA, 2009. ACM.

[11] Geoffrey Belter, Jeremy G. Siek, Ian Karlin, and E. R. Jessup. Automatic generation of tiled and parallel linear algebra routines. In In the Fifth International Workshop on Automatic Performance Tuning (iWAPT10), pages 1–15, Berkeley, California, June 2010.

[12] B.K. Bergen, M.G. Daniels, and P.M. Weber. A hybrid programming model for compressible gas dynamics using opencl. In Parallel Processing Workshops (ICPPW), 2010 39th International Conference on, pages 397–404, 2010.

[13] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and James Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In Proceedings of 11th International Conference on Supercomputing, pages 340–347, New York, NY, July 1997. ACM Press.

[14] L. S. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on (CDROM), pages 1–5, November 1996.

[15] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). ACM Transactions on Mathematical Software, 28(2):135–151, June 2002.

[16] Uday Bondhugula. Effective Automatic Parallelization and Optimization Using the Polyhedral Model. PhD thesis, The Ohio State University, August 2008.

[17] Uday Bondhugula. PLUTO an automatic loop nest parallelizer for multicores. http://pluto-compiler.sourceforge.net/, August 2010.

[18] 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'08/ETAPS'08: Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] François Broquedis, Jérôme Clet Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, pages 180–186, Pisa Italie, February 2010.

[20] Surendra Byna, Xian-He Sun, William Gropp, and Rajeev Thakur. Predicting memory-access cost based on data-access patterns. In Proceedings of the 2004 IEEE International Conference on Cluster Computing, pages 327–336, San Diego, CA, September 2004.

[21] Jonathan Carter, Leonid Oliker, and John Shalf. Performance evaluation of scientific applications on modern parallel vector systems. In High Performance Computing for Computational Science - VECPAR 2006, volume 4395 of Lecture Notes in Computer Science, pages 490–503. Springer Berlin / Heidelberg, May 2007.

[22] Texas Advanced Computing Center. GotoBLAS. http://www.tacc.utexas.edu/resources/software/#blas, 2007.

[23] Chun Chen, Jacqueline Chame, and Mary Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In CGO '05: Proceedings of the international symposium on Code generation and optimization, pages 111–122, Washington, DC, USA, March 2005. IEEE Computer Society.

[24] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David W. Walker, and R. Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. In PARA '95: Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, pages 107–114, London, UK, 1996. Springer-Verlag.

[25] A. T. Chronopoulos. s-step iterative methods for (non)symmetric (in)definite linear systems. SIAM J. Numer. Anal., 28:1776–1789, December 1991.

[26] Alain Darte. On the complexity of loop fusion. Parallel Computing, 26:1175–1193, July 2000.

[27] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Review, 51:129–159, February 2009.

[28] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, pages 1–12, April 2008.

[29] J. M. Dennis. Automated memory analysis: improving the design and implementation of iterative algorithms. PhD thesis, University of Colorado, Boulder, CO, July 2005.

[30] J. M. Dennis and E. R. Jessup. Applying automated memory analysis to improve iterative algorithms. SIAM Journal on Scientific Computing, 29(5):2210–2223, September 2007.

[31] J. Dongarra, D. Gannon, G. Fox, and K. Kenned. The impact of multicore on computational science software. CTWatch Quarterly, 3:3–10, 2007.

[32] J. Dongarra and R. Whaley. A user's guide to the BLACS v 1.0. Technical Report UT CS-95-281, LAPACK Working Note No. 94, University of Tennessee, Knoxville, TN, March 1995.

[33] J. J. Dongarra, J. Bunch, C. Moler, and G. Stewart. LINPACK Users' Guide. SIAM, Philadelphia, Pa., 1979.

[34] Jack Dongarra. Preface: Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard I. International Journal of High Performance Applications and Supercomputing, 16(1):1–111, Spring 2002.

[35] Jack Dongarra. Preface: Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard II. International Journal of High Performance Applications and Supercomputing, 16(2):115–199, Summer 2002.

[36] Jack J. Dongarra, Jeremy De Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software, 14(1):1–17, March 1988.

[37] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software, 16(1):1–17, March 1990.

[38] Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rude, and Christian Weiss. Cache optimization for structured and unstructured multigrid. Electronic Transactions on Numerical Analysis, 10:21–40, 2000.

[39] I. Duff, M. Heroux, and R. Pozo. An overview of the Sparse Basic Linear Algebra Subprograms: The new standard from the BLAS technical forum. ACM TOMS, 28(2):239–267, June 2002.

[40] Arkady Epshteyn, Mara Garzaran, Gerald DeJong, David Padua, Gang Ren, Xiaoming Li, Kamen Yotov, and Keshav Pingali. Analytic models and empirical search: A hybrid approach to code optimization. In Languages and Compilers for Parallel Computing, volume 4339 of Lecture Notes in Computer Science, pages 259–273. Springer Berlin / Heidelberg, 2006.

[41] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. Lecture Notes in Computer Science, 589:328–343, 1991.

[42] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. Proceedings of the IEEE, 93(2):216–231, February 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[43] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting, pages 97–104, Budapest, Hungary, September 2004.

[44] G. Gao, R. Olson, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, pages 281–295, New Haven, CT, Aug. 2004.

[45] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. Matrix Eigensystem Routines-EISPACK Guide Extension, volume 51. Springer-Verlag, New York, 1977.

[46] M. W. Gee, C. M. Siefert, J. J. Hu, R.S. Tuminaro, and M. G. Sala. ML 5.0 Smoothed Aggregation User's Guide. Technical Report SAND2006-2649, Sandia National Laboratories, May 2006.

[47] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In Proceedings of the 1997 ACM International Conference on Supercomputing, pages 317–324. ACM Press, July 1997.

[48] G. Golub and W. Kahan. Calculationg the singular values and pseudo-inverse of a matrix. Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis, 2(2):205–224, 1965.

[49] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. ACM Transactions on Mathematical Software, 34(3):25, May 2008.

[50] Khronos OpenCL Working Group. The OpenCL Specification Version 1.1. http://www.khronos.org/opencl, 2011.

[51] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. ACM Transactions on Mathematical Software, 27(4):422–455, December 2001.

[52] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, third edition, 2003.

[53] Vincente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. ACM Transactions on Mathematical Software (TOMS), 31(3):351–362, September 2005.

[54] Michael A. Heroux, Roscoe A. Barlett, Vicki E. Howell, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. ACM Transactions on Mathematical Software (TOMS), 31(3):397–423, September 2005.

[55] Gary W. Howell, James W. Demmel, Charles T. Fulton, Sven Hammarling, and Karen Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. ACM Transactions on Mathematical Software (TOMS), 34(3), May 2008.

[56] IBM. Engineering Scientific Subroutine Library. http://www-03.ibm.com/systems/p/software/essl/index.html, 2008.

[57] Intel. Intel Math Kernel Library. http://www.intel.com/cd/software/products/asmo-na/eng/307757, 2007.

[58] Intel. Intel Compilers and Libraries - Intel Software Network. http://software.intel.com/en-us/articles/intel-compilers, 2010.

[59] Raj Jain. The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation and Modeling. Wiley, New York, 1991.

[60] Elizabeth R. Jessup, Ian Karlin, Erik Silkensen, Geoffrey Belter, and Jeremy Siek. Understanding memory effects in the automated generation of optimized matrix algebra kernels. Procedia Computer Science, 1(1):1867 – 1875, May 2010.

[61] I. Karlin. Memory analysis and tuning of composed linear algebra kernels. In Proceedings of Colorado Celebration of Women in Computing, pages 1–5, Boulder, CO, April 2008.

[62] I. Karlin, E. Silkensen, E. R. Jessup, G. Belter, T. Nelson, P. Zelinsky, and J. G. Siek. A statistical approach to reducing an optimization search space. In Proceedings of Colorado Celebration of Women in Computing, pages 1–5, Golden, CO, November 2010.

[63] Ian Karlin and Jonathan Hu. Implementing and profiling of a variable block matrix-matrix multiply in ML. Technical Report SAND 2007-7977, Sandia National Laboratories, December 2007.

[64] Ian Karlin and Jonathan Hu. Overview and performance analysis of the epetra/OSKI matrix class in trilinos. Technical Report SAND2008-8257P, Sandia National Laboratories, December 2008.

[65] Ian Karlin, Elizabeth R. Jessup, Geoffrey Belter, and Jeremy Siek. Parallel memory prediction for fused linear algebra kernels. In Proceedings of 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10), pages 1–8, New Orleans, LA, November 2010.

[66] A D. K. Kaushik, B D. E. Keyes, and B. F. Smith D. Toward realistic performance bounds for implicit cfd codes. In Proceedings of Parallel CFD99, pages 233–240. Elsevier, 1999.

[67] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Kriszti? Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage current: Moore's law meets static power. Computer, 36(12):68–75, December 2003.

[68] D. Kincaid and W. Cheney. Numerical Analysis: Mathematics of Scientific Computing. Brooks/Cole, third edition, 2002.

[69] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. J. ACM, 46:604–632, September 1999.

[70] Monica S. Lam, Edward E. Rothber, 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, pages 63–74, Palo Alto, CA, Apr. 1991.

[71] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. ACM Transactions on Mathematical Software, 5(3):308–323, September 1979.

[72] Robert Michael Lewis, Virginia Torczon, and Michael W. Trosset. Direct search methods: then and now. Journal of Computational and Applied Mathematics, 124(1-2):191 – 207, December 2000.

[73] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pages 19–25, December 1995.

[74] Lois Curfman McInnes, Jorge Moré, Todd Munson, and Jason Sarich. TAO user manual (revision 1.10.1). Technical Report ANL/MCS-TM-242-Revision 1.10.1, Mathematics and Computer Science Division, Argonne National Laboratory, July 2010.

[75] J.C. Meyer and A.C. Elster. Performance modeling of heterogeneous systems. In Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, pages 1 –4, april 2010.

[76] Frank Mueller. A library implementation of POSIX threads under UNIX. In In Proceedings of the USENIX Conference, pages 29–41, January 1993.

[77] Thomas Nelson. Personal communication, November 2010.

[78] Netlib. BLAS. http://www.netlib.org/blas/index/html, 2008.

[79] Netlib. Sparse BLAS. http://www.netlib.org/sparse-blas/index.html, 2008.

[80] NIST. Sparse BLAS. http://math.nist.gov/spblas, 2008.

[81] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeon, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM. IEEE Micro, pages 34–44, March/April 1997.

[82] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–11, Washington, DC, USA, November 2010. IEEE Computer Society.

[83] Madhan Premkumar. Parallelization of cache efficient BLAS 2.5 operator GEMVT. Master's thesis, Florida Institute of Technology, Melbourne, FL, May 2005.

[84] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation", 93(2):232– 275, February 2005.

[85] Apan Qasem. Automatic Tuning of Scientific Applications. PhD thesis, Rice University, July 2007.

[86] Apan Qasem and Ken Kennedy. A cache-conscious profitability model for empirical tuning of loop fusion. In Eduard Ayguad, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan, editors, LCPC, volume 4339 of Lecture Notes in Computer Science, pages 106–120. Springer, 2006.

[87] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, pages 38–49, Montreal, Quebec, Canada, May 1998.

[88] Gabriel Rivera and Chau-Wen Tseng. Tiling optimizations for 3D scientific computations. In Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), pages 1–23, Washington, DC, USA, November 2000. IEEE Computer Society.

[89] M. Sala, K. Stanley, and M. Heroux. Amesos: A set of general interfaces of sparse direct solver libraries. 4699:976–985, 2007.

[90] Sandia National Laboratories. Epetra - Home.
http://trilinos.sandia.gov/packages/epetra/index.html, 2008.

[91] Sandia National Laboratories. Sacado - Home.
http://trilinos.sandia.gov/packages/sacado/index.html, 2008.

[92] Jeremy G. Siek, Ian Karlin, and E. R. Jessup. Build to order linear algebra kernels. In Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL 2008), pages 1–8, Miami, FL, April 2008.

[93] Horst Simon, Leonid Oliker, Andrew Canning, Jonathan Carter, Stephane Ethier, and John Shalf. Evaluation of leading scalar and vector architectures for scientific computations. Technical Report LBNL-55291, Lawrence Berkeley National Laboratory, 2004.

[94] B. T Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Kelma, and C. B. Moler. Matrix Eigensystem Routines EISPACK Guide, volume 6. Springer-Verlag, New York, 2nd edition, 1976.

[95] B. Spencer Jr., T. Finholt, I. Foster, C. Kesselman, C. Beldica, J. Futrelle, S. Gullapalli, P. Hubbard, L. Liming, D. Marcusiu, L. Pearlman, C. Severance, and G. Yang. NEESgrid: A distributed collaboratory for advanced earthquake engineering experiment and simulation. In 13th World Conference on Earthquake Engineering, Vancouver, B.C, Canada, Aug 2004. Paper NO. 1674.

[96] Sun. Sun Performance Library. http://developers.sun.com/sunstudio/overview/topics/perflib_index.html, 2008.

[97] G. Videl. Efficient simulation of one dimensional quantum many-body systems. In Physical Review Letters, 93(4):1–4, July 2004.

[98] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. Journal of Physics: Conference Series, 16:521–530, June 2005.

[99] Richard Vuduc. Personal communication, July 2008.

[100] Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In Proceedings of the IEEE/ACM Conference on Supercomputing, pages 1–35, Baltimore, MD, November 2002.

[101] Richard Vuduc, Attila Gyulassy, James W. Demmel, and Katherine A. Yelick. Memory hierarchy optimizations and performance bounds for sparse A^TAx. In ICCS 2003: Workshop on Parallel Linear Algebra, Melbourne, Australia, June 2003.

[102] Richard W. Vuduc. Automatic performance tuning of sparse matrix kernels. PhD thesis, University of California, Berkeley, CA, USA, January 2004.

[103] W. Wang and D. P. O'Leary. Adaptive use of iterative methods in predictor-corrector interior point methods for linear programming. 25:387–406, September 2000.

[104] Shlomo Weiss and James E. Smith. Study of scalar compilation techniques for pipelined supercomputers. ACM Transactions on Mathematical Software (TOMS), 16(3):223–245, September 1990.

[105] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In Proceedings of 1998 ACM/IEEE Conference on Supercomputing (CDROM), pages 1–27, Washington DC, November 1998. IEEE Computer Society.

[106] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Parallel Computing, 35(3):178 – 194, 2009.

[107] M. Xue, K. K. Droegemeier, and V. Wong. The advanced regional prediction system (ARPS) - a multiscale nonhydrostatic atmospheric simulation and prediction model. Part I: model dynamics and verification. Meteorology and Atmospheric Physics, 75:161–193, December 2000.

[108] 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.

[109] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Think globally, search locally. In ICS '05: Proceedings of the 19th annual international conference on Supercomputing, pages 141–150, New York, NY, USA, June 2005. ACM.