# Parallel Memory Prediction for Fused Linear Algebra Kernels

Ian Karlin
University of Colorado, Boulder
Dept of Computer Science
ian.karlin@colorado.edu

Elizabeth R. Jessup
University of Colorado, Boulder
Dept of Computer Science
jessup@colorado.edu

Geoffrey Belter
University of Colorado, Boulder
Dept of ECEE
belter@colorado.edu

Jeremy G. Siek
University of Colorado, Boulder
Dept of ECEE
jeremy.siek@colorado.edu

## ABSTRACT

The performance of many scientific programs is limited by data movement. Loop fusion is one optimization used to increase the speed of memory bound operations. To automate loop fusion for matrix computations, we developed the Build to Order (BTO) compiler. Within BTO, an analytic memory model efficiently and accurately reduces the number of serial loop fusion options considered. In this paper, we extend the model to shared memory parallel machines. We detail the differences between parallel and serial memory use and runtime prediction and explain the changes made to include parallel machines in the model. Analysis of the parallel model's predictions show that when it is included in BTO it will be able to trim the search space of considered routines.

## Keywords

Memory Modeling, Parallel Processing, Auto-tuning

## 1. INTRODUCTION

Data movement from memory to the processor limits the performance of many scientific programs [1]. Often these programs perform few floating-point computations per memory read. To counteract this problem, tuning techniques are used to reduce the amount of memory traffic required to perform calculations [2, 3, 12, 19]. Loop fusion is one such optimization technique that reduces data movement by combining multiple loops [16].

When loop fusion is applied to memory bound linear algebra operations, large reductions in runtime often result [6, 7, 23, 27]. Creating efficient fused routines is important since the kernel being fused is often a large portion of the overall routine runtime [19]. However, one cannot simply fuse all possible loops and expect good performance because fusion interacts in both positive and negative ways with the memory hierarchy and other optimizations [20, 21]. Therefore, the optimal amount of loop fusion and optimizations with which to combine it is not always obvious, and without exploring the whole fusion search space, it is possible to produce a sub-optimal routine. However, the problem of finding the optimal amount of loop fusion is NP-Complete [11], and, therefore, it is not always feasible to empirically test all possible amounts of fusion.

To solve this problem, we created a domain-specific language and a compiler that uses it, Build to Order (BTO) [3, 25]. The compiler takes in a annotated subset of MATLAB and produces optimized kernels in C. The initial optimizations included were two forms of loop fusion. The addition of data partitioning enables cache blocking and the creation of shared memory parallel codes [4]. BTO enumerates the entire search space of potentially profitable loop fusion combinations that access common data. Those operations are then tested using a hybrid analytical/empirical testing methodology. The compiler analyzes the runtimes of the fused routines to identify 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 this paper, we first extend our analytic model to shared memory parallel machines. The steps required to add parallel capabilities include altering data structures and adding new capabilities to memory miss and runtime prediction functions. To make it easier to run the BTO compiler, we automate the process of determining the architectural specifications of a machine and the amount of useable bandwidth between caches and memory and the processor.

This paper is organized as follows. In Section 2, we present related work. In Section 3, we describe how data structures are altered to store the necessary information to model parallel machines. Also, included in this section is an explanation of how we automate the process of determining machine specifications. We then detail our serial memory traffic prediction algorithm and explain how we updated it to account for parallel machines in Section 4. Assumptions made in its design and why they were made are described in this section. In Section 5, we show how we convert memory traffic


*PMBS10* New Orleans, Louisiana USA

estimates into runtime predictions. We describe our serial method, the multiple execution patterns that exist in parallel and how to model them. Section 6 contains a comparison of memory traffic predictions and runtime predictions to actual values. Based on these results, we believe that the parallel model will be effective once integrated into the BTO system in analyzing the loop fusion search spaces for shared memory parallel machines. We conclude the paper with Section 7 presenting conclusions and future work.

# 2. RELATED WORK

Other work on loop fusion, memory and performance modeling, automating the generation of linear algebra routines, and hybrid search techniques provides us inspiration. In this section, we detail the contributions of other authors and show how our project and model both differ and build off of their work.

***Loop Fusion***

Loop fusion is an optimization that combines multiple loops into a single loop. When the combined loops access the same data, temporal locality of the accesses can improve, resulting in performance benefits. Applying loop fusion to memory-bound linear algebra routines often results in significant runtime reductions [19, 27]. However, fusing loops does not always increase routine performance. For example, too much fusion can cause register spill [20] or data to not fit within cache [21]. Therefore, when fusing loops it is important to consider the target architecture along with the legality and potential benefits of fusion.

***Memory and Performance Predictions Methods***

To estimate performance without running the program, analytic models can be used to predict cache misses, register misses, or runtime. Different approaches trade speed and accuracy in various ways to produce estimates for diverse problems. One approach that emphasizes extreme accuracy is the Cache Miss Equations [17]. This formulation of equations to calculate reuse distances results in near exact estimates of cache behavior but is expensive to compute. We also calculate reuse distances in our model but only for machine and routine parameters that are most distinguishing to fused routine performance. Ferrante et al. [14] use a model that provides capacity based estimates of the number of cache lines accessed for perfectly nested loops to find the best ordering of loops for matrix multiplication. We perform capacity based estimates in our model but also handle arbitrary loop nests. Rivera and Tseng [24] model conflict misses to predict the profitability of array padding. We use empirical testing to determine details like conflict misses and tiling to reduce them.

Byna et al. [9] show that the runtime performance of memory-bound routines can be predicted from memory traffic predictions and data access costs. We also convert memory access predictions into runtime predictions and extend them to parallel architectures. Yotov et al. [29] use an analytic model that performs comparably well to empirical search in approximately half the time for matrix-matrix multiplication.

***Automatic Generation of Linear Algebra Routines***

Many authors use self-optimizing libraries to produce efficient linear algebra routines. ATLAS [28] and PHiPAC [5] optimize matrix-matrix multiplication using empirically guided compilation. OSKI [26] performs install time machine benchmarking and runtime optimizations for sparse matrices, including three fused kernels. Our compiler allows user input of linear algebra routines allowing for the creation of arbitrary kernels with fewer restrictions. FLAME [18], which focuses on the derivation of correct linear algebra algorithms, partially automates the process of generating kernels. Our compiler is fully automatic.

***Hybrid Search Techniques***

Recent efforts using hybrid search methods that combine analytic modeling and empirical testing have been successful. Chen et al. [10] first use analytic techniques to enumerate combinations of prefetching, loop unrolling, register and loop tiling, and copy optimization. They then empirically test these optimization strategies, choosing the best. Epshteyn et al. [13] tune matrix multiplication using an explanation based learning algorithm for loop tiling decisions. The learning algorithm adapts their analytic model based on empirical results. Yotov et al. [30] propose using analytic analysis at a global level and then empirical search to fine-tune performance. We also employ a hybrid search approach using analytical modeling and empirical search.

***General Loop Fusion Tools***

Two loop fusion specific approaches that work on general purpose languages are PLUTO [7], which uses the polyhedral model to generate fused and parallel codes and Qasem's method which performs fusion on Fortran codes [23]. PLUTO uses heuristics to narrow the search space, while Qasem uses a memory model to guide empirical search. Both limit search space exploration and prediction efforts because they target general purpose languages. Therefore, both of these approaches may not find the globally optimal solution. For a more restricted domain, such as linear algebra, it is feasible to analyze the entire search space, as we show in [3].

# 3. DATA STRUCTURES AND MACHINE REPRESENTATION

The input into our memory model is an Abstract Syntax Tree (AST) representation of a calculation and a machine structure. In this section, we present both as used for serial computations and then describe the information we add to them for parallel computations. We also present the methods we use to automatically determine hardware specifications for a target system.

## 3.1 Calculation Representation

An AST is used to represent a calculation's input into the model. For serial predictions, the AST contains nodes that represent loops and statements and leaves that represent variables. A loop consists of the name of its iterate and number of iterations run, while a variable contains the names of all iterates used to access it.

For routines run in parallel, we add one additional field to identify variables that store partial sums. Partial sums occur when each core computes a portion of a result stored separately and then later combined. Each partial sum is stored in memory increasing the amount of data in shared caches. The same field indicates when multiple cores access different parts of the same data structure simultaneously, also resulting in extra data being present in shared caches.

## 3.2 Machine Representation

To represent a serial machine, we use one data structure to model each memory component (caches, TLBs and registers), its size and the amount of bandwidth between the next largest memory structure and the processor. Bandwidth of each bus to the next largest component is stored there because it represents the number of misses to the component at hand.

To represent a parallel machine, we convert to a hierarchical machine representation, where representations of all memory components accessed by the same number of cores are on the same level. Stored at each level is how many cores share caches or buses to the cache. The levels mimic the relationships between memory components, with those farthest from processing cores at the top of the representation. Identical memory components are compressed to a single representation.

## 3.3 Automatic Machine Feature Detection

To determine machine specifications including memory structure sizes, bandwidths and machine topology we use two available programs and one of our own. We call these programs automatically when BTO is installed, to ease the use of BTO.

### 3.3.1 Finding Cache Specifications

To determine the number of caches and their sizes, we use hwloc [8]. Hwloc is a subproject of OpenMPI [15] that determines the memory hierarchy and processor layout of most modern computers including the number of cores and sockets a machine contains. Also determined is the layout of cores and caches on sockets, including which cores share which caches. Hwloc works on operating systems as varied as Linux, OS X, Microsoft Windows and Solaris.

### 3.3.2 Determining Number of Registers

To determine the number of registers accessible by the compiler on a system, we use the following procedure. A test code produces $n + 1$ programs with bodies as follows:

```
register int a0;
register int a1;
       :
       :
register int an;
a0 = 03456789;
a1 = 13456789;
       :
       :
an = n3456789;
a0 = a0 + a1 + ... + an;
```

Each is compiled to assembly code and all lines containing 3456789 are counted and pulled out of the file for analysis. The 3456789 pattern is used to identify lines where stores occur. Example assembly for a test code compiled for regular registers in 32 bit mode with n = 8 on an x86 machine is as follows:

```
         :
         :
movl    $53456789, %esi
movl    $63456789, %edi
movl    $73456789, -16(%ebp)
movl    $83456789, -12(%ebp)
```

Indirect addressing occurs when registers are surrounded by parenthesizes indicating reads from memory. All lines not containing indirect addressing indicate that a variable $aj$ is stored in a register. Therefore, to come up with the number of registers used for a program, each line that contains indirect addressing is removed from the count. The program is run until three codes in a row produce the same register count, which is used to represent the number of registers on a machine. The procedure works in 32 bit or 64 bit mode for regular and vector registers.

### 3.3.3 Measuring Usable Bandwidth

To determine the amount of bandwidth available from a memory structure to the processor, we use the STREAM TRIAD benchmark [22]. 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, requiring two loads and a store to perform one multiplication and one addition. 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, a data set that is the average of the size of the cache being profiled and the next smaller cache is used. A data set three times the largest cache size is used to profile the bandwidth from main memory to the processor. The bandwidth from the smallest cache to the processor is stored in the machine structure with the registers. The bandwidth from the second smallest cache to the processor is stored with the smallest cache and so on until the bandwidth from memory to the processor is stored with the largest cache. Therefore, each memory structure stores the bandwidth that misses to it uses.

# 4. MEMORY TRAFFIC PREDICTIONS

The first phase of our model is memory traffic prediction where a calculation is analyzed to determine how many reads will occur from each memory structure. In this section, we discuss assumptions made in the model and why they were made. We then present our serial prediction approach. Next, we explain how architectural features and data distribution on parallel machines impact memory traffic. Finally, we detail the changes made to our serial memory traffic predictions to incorporate these parallel factors.

## 4.1 Assumptions

To increase the speed of our model at a small accuracy cost, we restrict it to computing the most distinguishing factors. We assume all data are accessed consecutively because BTO only produces consecutive access patterns. Thus, latency bound reads are rare and so latency is not modeled. We treat all caches (caches refers to both caches and TLBs) the same and model registers similarly with a few modifications. All memory structures are assumed to be fully associative and all caches to use a least recently used replacement policy. Empirical testing detects the differences in runtime caused by conflict misses. We also assume memory structures are "warm" at the start of kernel execution.

## 4.2 Serial Predictions

The model uses reuse distances, which measure the amount of unique data touched between two accesses to a data element. Reuse distances are calculated for the first element

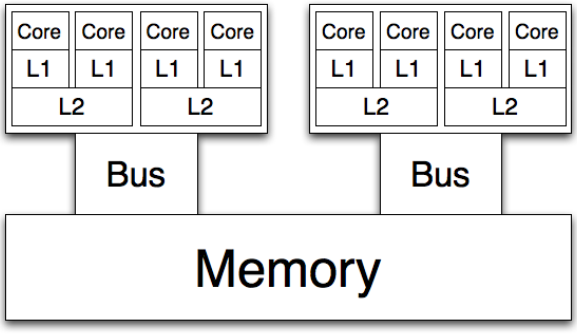


**Figure 1: Dual socket quad-core Intel Clovertown**

of an array and used for all elements of that array. In practice, reuse distances throughout an array can change on the order of the size of one dimension of the array. However, arrays are equally likely to have increasing or decreasing reuse distances as accesses move further away from the start of the array. Sacrificing accuracy here allows for significant runtime speedups of the model while still allowing BTO to distinguish between routines with large memory cost differences accurately. Empirical testing detects conflict misses and, in practice, they only matter near cache boundaries.

The model compares the calculated reuse distance of an array to all cache sizes. For each cache where the reuse distance is larger than the cache size, all accesses to that data structure are assumed to occur from the next largest memory structure. Therefore, reads to one data structure are charged to multiple memory structures. The number of reads from the next largest memory structure is stored in the AST.

Compiler algorithms determine when data are stored in registers. Instead of using reuse distance and assuming least recently used replacement, our model allocates registers in the following order of priority. The iterate of the inner most loop always resides in a register. Then variables accessed multiple times within an inner loop are stored. Finally, variables where subsequent interactions of the current loop do not change which data are accessed are stored.

### 4.3 Architectural Features and Data Distribution on Parallel Machines

Most shared memory parallel systems have multiple sockets, each containing a processor with multiple cores. For example, the dual socket quad-core Intel Clovertown shown in Figure 1 contains four L2 caches each shared by two cores and an L1 cache per core that is not shared. The additional caches increase the amount of data that can be stored close to the processors. However, when caches are shared among cores, 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.

To take advantage of the additional processors, work must be divided among all of their cores. Doing so adds a number of potential memory impacts when executing a calculation. It can result in extra data being stored in shared caches. For example, when computing a matrix-vector multiply using axpys, each core produces a partial result of the whole vector. These partial results need to be stored separately and summed. Also, extra data reside in cache when cores that share a cache access different parts of the same data structure. The increased data stored in caches can result in added misses, leading to increased reads from slower memory structures. However, if inner products are used to compute the matrix-vector multiply, the extra data needed in cache per core is only that needed to store the two columns of the matrix.

### 4.4 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 the results of 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. To determine the memory structure from which reads to each data structure occur, reuse distances are compared to memory structure sizes.

## 5. RUNTIME PREDICTION

The second phase of our memory model converts memory traffic estimates into runtime predictions. In this section, we explain our method for single processor systems. Next we discuss how the architectural features of parallel machines affects performance in parallel. Included in this discussion is how routine execution affects performance. Then we explain how we changed our serial runtime prediction approach to account for parallel architectures.

### 5.1 Serial Runtime Predictions

To turn memory traffic predictions into runtime predictions, we use the number of misses to each memory structure and the bandwidth between each structure and the processor. We start our analysis with innermost loops where the first step is determining which memory structure limits performance. Accesses to data in registers are assumed to have no cost. For all other memory structures, the amount of data accessed from that structure is divided by the bandwidth between the structure and the processor, resulting in the cost of moving data between that memory structure and the processor. The maximum of these costs represents the cost of running the loop and the memory structure that produced it is the one that limits the performance of that loop.

For loops that contain other loops, costs are calculated in

two ways. The method used for innermost loops is applied to all memory accesses that occur within a loop. Also, for outer loops the cost of running all inner loops plus the cost of any instructions in the current loop are summed. The maximum of these two costs represents the cost of running the outer loop. This process is repeated for the whole AST until the root is reached. The cost of executing the root loop is the cost of executing the routine.

## 5.2 Architectural Features and Routine Execution Patterns

On parallel shared memory machines, the number of buses from memory and caches to the processor increases when compared to serial machines. For example, the Intel Clovertown in Figure 1 has two buses from memory, one per socket, and one bus per core from each level of cache. These added buses increase the overall amount of bandwidth available to perform calculations. However, when buses are shared between cores, the amount of memory bandwidth available per core decreases and contention for the bus can result.

Another observation from our Clovertown system that is also true for many other modern processors is that bandwidth to the processor increases by as much or more the closer a memory structure is to the processor. In the Clovertown example, memory to processor bandwidth is increased by a factor of two when using the whole system, while the bandwidth from both caches to the processor is increased by a factor of eight as compared to using one core in serial. The non-uniform increase in bandwidths can cause serial calculations that are bound on data traffic from a cache to become bound on data traffic from memory when executed in parallel.

Accounting for how a routine can be executed is also necessary on parallel machines. Due to non-deterministic execution patterns, different amounts of contention for hardware resources among cores can occur. 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 in Figure 2. The unfused calculation is on the left and the fused calculation on the right. For the fused routine we present two execution paths that are the extremes and we use them as performance bounds. In actuality, a processor can switch between execution patterns unless the code was written to enforce one or the other.

To demonstrate these two possibilities, we describe two ways the calculation $b = AA^Tx$ can be performed. 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 outer loop iterations between the cores.

For the first execution pattern, each core executes the same inner loop, $t = A^T * x$ in Figure 2, 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 loop are now 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 $b = A * t$. Since only one core is reading data from main memory there is no bus contention. Then each core performs the other calculation.

If moving data from memory is the limiting factor for

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

**Figure 2: Fusing $b = AA^Tx$**

routine performance, then the first algorithm executes more slowly than the second. The difference 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 performing it on one core while the more expensive $t = A^Tx$ runs on the other core. However, if data movement from cache limits performance, the performance of both algorithms is the same since performing both parts of $b = At$ at the same time does not create bus contention because cores performing the calculation use separate buses to access cache.

## 5.3 Parallel Runtime Prediction

For converting memory traffic estimates to runtime predictions on parallel machines, both possible 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 the preceding section 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 much data is 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, accounting for the increase in bandwidth created by multiple buses. The same algorithm is used to traverse the AST since its traversal scheme also in parallel implies that all cores execute both inner loops concurrently. To account for parallel structures, bandwidths are multiplied by the number of buses from a memory structure to cores. For our example Clovertown machine in Figure 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. Total misses for all cores to a memory structure for all inner loops are 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. 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. RESULTS

To test the accuracy and usefulness of our parallel model to predict performance in BTO, we ran a series of tests. Accuracy is measured by how closely model predictions of memory traffic and runtimes match measured values obtained from timings and hardware counters. Usefulness in BTO is measured by how well the model distinguishes actual performance differences between versions of the same routine. Tests were run on a dual-core Core 2 duo system with a 2.4 GHz processor, 32 KB of L1 cache, 4 MB of L2 cache and 4 GB of memory for fused and unfused $b = AA^Tx$ and $r = A^Tx, s = Ay$ shown in Figures 2 and 3. In this section, we present the results of these tests.

```
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: Fused and unfused $Ax = r, A^Ty = s$**

## 6.1 Memory Predictions

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 4 and 5, we show the number of 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 and few misses 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. 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.

## 6.2 Runtime Predictions

The second step in evaluating the accuracy of our parallel memory model was comparing runtime predictions to the actual routine runtimes. Our runtime estimates were not accurate in estimating actual performance, but they 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 our model is useful in determining which versions produced by BTO will perform the best. Inexact runtime predictions may result from inaccurate profiling of the bandwidth available in parallel. Using a more accurate benchmarking procedure would help with predictions. However, the relative accuracy of our predictions demonstrate the model will accurately and efficiently trim the search space of versions of the same routine considered within BTO.

# 7. CONCLUSIONS AND FUTURE WORK

In this paper, we present how we improve our serial memory model to handle shared memory parallel machines. We detail the design decisions we made and the reasons for them. We demonstrate the accuracy of our parallel model and show that it works comparably well to our serial model. From these results we believe that when we integrate the parallel model within the BTO compiler it will perform as well as the serial model did in trimming the search space of parallel routines.

The model is not perfect and in this paper we identified two areas where we can improve the model to increase its accuracy and usefulness as a prediction tool. We also will explore whether distributing the workload of fused computations in other ways among processors improves performance.

# 8. ACKNOWLEDGEMENTS

The work of E. R. Jessup and I. Karlin is supported by National Science Foundation award CCF 0830458. The work of J. G. Siek and G. Belter is supported by CAREER: Bridging the Gap Between Prototyping and Production. NSF Grant Number 0846121.

# 9. REFERENCES

[1] 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. *Supercomputing, ACM/IEEE 1999 Conference*, pages 69–81, Nov 1999.

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

[3] G. Belter, E. R. Jessup, I. Karlin, and J. 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.

[4] G. Belter, J. G. Siek, I. Karlin, and E. R. Jessup. Automatic generation of tiled and parallel linear algebra routines. In *In the Fifth International Workshop on Automatic Performance Tuning (iWAPT'10)*, pages 1–15, June 2010.

[5] J. Bilmes, K. Asanovic, C.-W. Chin, and J. 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, 1997. ACM Press.

[6] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman,

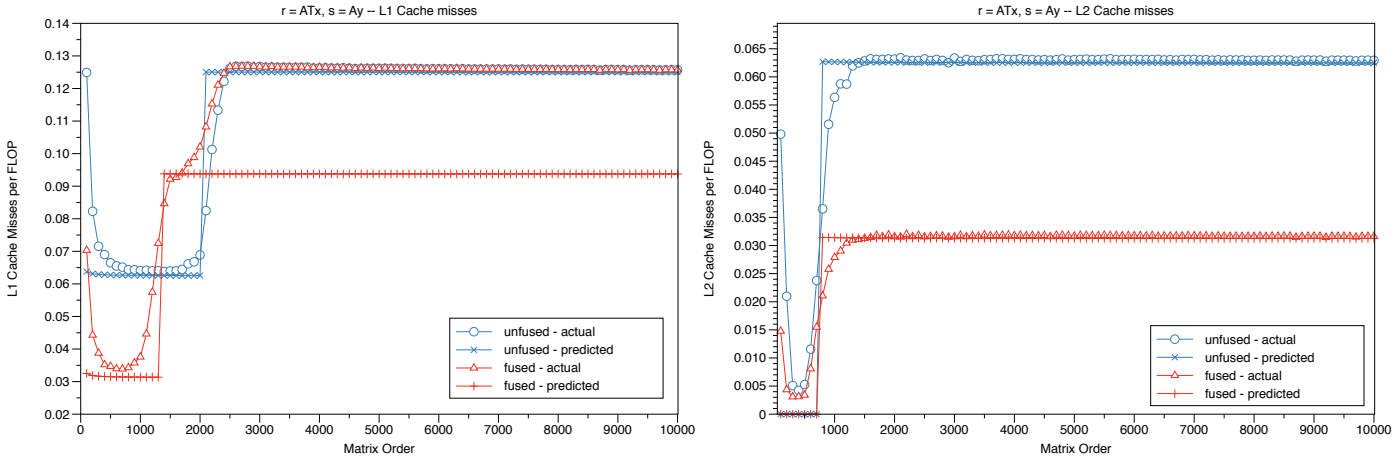


**Figure 4: L1 and L2 predicted and actual misses for $r = A^T x, s = Ay$.**

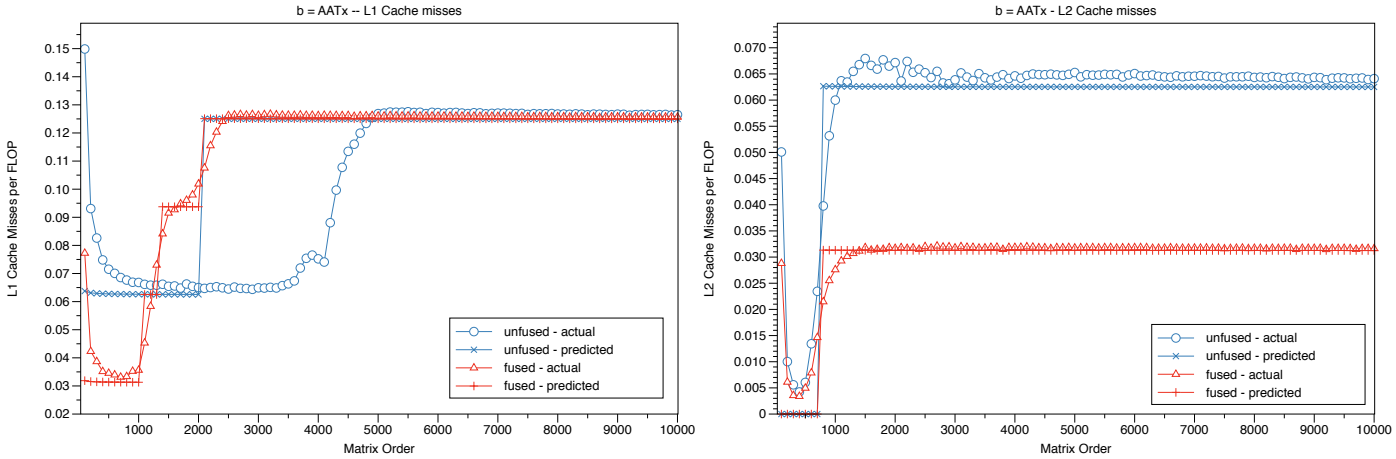


**Figure 5: L1 and L2 predicted and actual misses for $b = AA^T x$.**

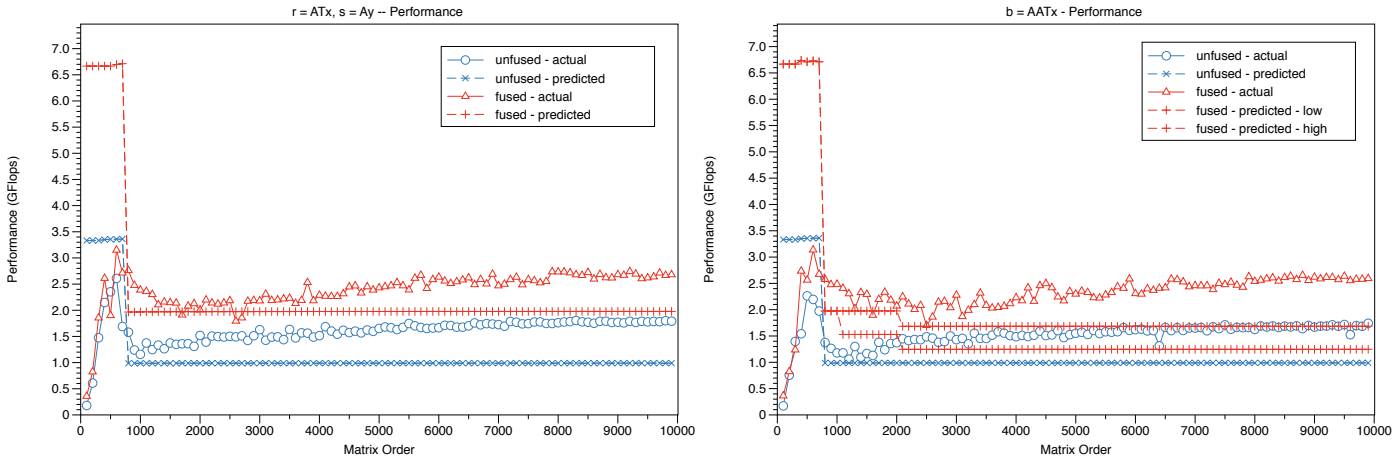


**Figure 6: Actual and predicted runtimes for $r = A^T x, s = Ay$ and $b = AA^T x$.**

A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[7] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. 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.

[8] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. 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*, Pisa, Italy, 02 2010.

[9] S. Byna, X.-H. Sun, W. Gropp, and R. 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, 2004.

[10] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society.

[11] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26:149–157, 1999.

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

[13] A. Epshteyn, M. Garzaran, G. DeJong, D. Padua, G. Ren, X. Li, K. Yotov, and K. 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.

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

[15] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. 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.

[16] 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.

[17] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21:703–746, 1999.

[18] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.*, 27(4):422–455, 2001.

[19] G. W. Howell, J. W. Demmel, C. T. Fulton, S. Hammarling, and K. Marmol. Cache efficient bidiagonalization using BLAS 2.5 operators. *ACM Transactions on Mathematical Software*, 34(3):14:1–14:33, 2008.

[20] E. R. Jessup, I. Karlin, E. Silkensen, G. Belter, and J. Siek. Understanding memory effects in the automated generation of optimized matrix algebra kernels. *Procedia Computer Science*, 1(1):1867 – 1875, 2010.

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

[22] J. 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, Dec. 1995.

[23] A. Qasem. *Automatic Tuning of Scientific Applications*. PhD thesis, Rice University, July 2007.

[24] G. Rivera and C.-W. 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, 1998.

[25] J. G. Siek, I. 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.

[26] 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.

[27] R. Vuduc, A. Gyulassy, J. W. Demmel, and K. A. Yelick. Memory hierarchy optimizations and performance bounds for sparse $A^T Ax$. In *ICCS 2003: Workshop on Parallel Linear Algebra*, Melbourne, Australia, June 2003.

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

[29] K. Yotov, X. Li, G. Ren, M. 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, February 2005.

[30] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 141–150, New York, NY, USA, 2005. ACM.