A Comparison of Empirical and Model-driven Optimization

Kamen Yotov[2]
kyotov@cs.cornell.edu

Xiaoming Li[1]
xli15@uiuc.edu

Gang Ren[1]
gangren@students.uiuc.edu

Michael Cibulskis[1]
cibulski@uiuc.edu

Gerald DeJong[1]
dejong@cs.uiuc.edu

Maria Garzaran[1]
garzaran@cs.uiuc.edu

David Padua[1]
padua@uiuc.edu

Keshav Pingali[2]
pingali@cs.cornell.edu

Paul Stodghill[2]
stodghil@cs.cornell.edu

Peng Wu[3]
pengwu@us.ibm.com

[1]University of Illinois at Urbana-Champaign [2]Cornell University [3]IBM T.J. Watson Research Center

ABSTRACT

Empirical program optimizers estimate the values of key optimization parameters by generating different program versions and running them on the actual hardware to determine which values give the best performance. In contrast, conventional compilers use models of programs and machines to choose these parameters. It is widely believed that model-driven optimization does not compete with empirical optimization, but few quantitative comparisons have been done to date. To make such a comparison, we replaced the empirical optimization engine in ATLAS (a system for generating a dense numerical linear algebra library called the BLAS) with a model-driven optimization engine that used detailed models to estimate values for optimization parameters, and then measured the relative performance of the two systems on three different hardware platforms. Our experiments show that model-driven optimization can be surprisingly effective, and can generate code whose performance is comparable to that of code generated by empirical optimizers for the BLAS.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *compilers, code generation, optimization*; I.2.2 [**Artificial Intelligence**]: Automatic programming – *program transformation*; G.4 [**Mathematical Software**]

General Terms

Algorithms, Measurement, Performance, Experimentation.

Keywords

Compilers, Memory hierarchy, Tiling, Blocking, Unrolling, Program transformation, Code generation, Empirical optimization, Model-driven optimization, BLAS

This work was supported by NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-012140

1. INTRODUCTION

The essential form of knowledge is nothing but a representation of truth:
for the truth of being and the truth of knowing are one,
differing no more than the direct beam and the beam reflected.
Francis Bacon, Advancement of Learning (1605)

High-level program transformations such as loop tiling, loop unrolling, and software pipelining are critical for compiling efficient code for modern architectures. Many of these transformations have numerical parameters whose values must be chosen carefully to obtain optimal performance. For example, to apply loop unrolling, it is necessary to determine how many times the loop must be unrolled; too little unrolling may result in inefficient use of processor resources while too much unrolling may cause instruction cache overflow, or register spills to memory.

To compute good values for transformation parameters, most compilers use simple architectural models that are tractable abstractions of the complex hardware of modern computers. When applying loop tiling for example, tile sizes are usually determined by assuming that the cache is fully associative, whereas most hardware caches have limited set-associativity and use a pseudo-LRU replacement policy.

Although there is a substantial body of work on restructuring compilers, it is fair to say that even for a simple kernel like matrix multiplication, most current compilers do not generate code that can compete with hand-written code in efficiency. To circumvent this difficulty, some library writers are using *empirical optimization* to generate highly tuned libraries automatically. Well-known library generators that employ empirical optimization are ATLAS [18] which generates highly tuned Basic Linear Algebra Subroutine (BLAS), FFTW [8] and SPIRAL [21] which generate FFT libraries. To choose a good tile size for a loop, a system that uses empirical optimization generates multiple versions of the tiled loop, runs all of them on the actual machine, and selects the tile size that results in the best performance. Simple architectural models can be useful to prune the size of the search space, but there is no need for precise models. These library generators produce better code on a wide range of architectures than native compilers using model-driven optimization.

Why are current compilers unable to transform a simple matrix multiplication loop into code that performs as well as the code generated by ATLAS? One possibility is that compilers are at a disadvantage because they are general-purpose and must be able to optimize any program, whereas ATLAS is a library generator that can focus on a particular problem domain. The trouble with this argument is that the problem domain of ATLAS is dense numerical linear algebra, which is precisely the area that has been studied most intensely by the compiler community! Another possibility is that systems like ATLAS are performing certain optimizations that compilers do not know about. Yet another possibility is that these systems incorporate the same optimizations as compilers do, but perform them in a different order (the so-called "phase-ordering problem"). Finally, if phase-ordering is not an issue, perhaps the architectural models used by compilers are overly simplistic compared to the complex hardware of modern computers, so they are unable to estimate optimal transformation parameters accurately. To the best of our knowledge, no studies exist to provide clear answers to these questions.

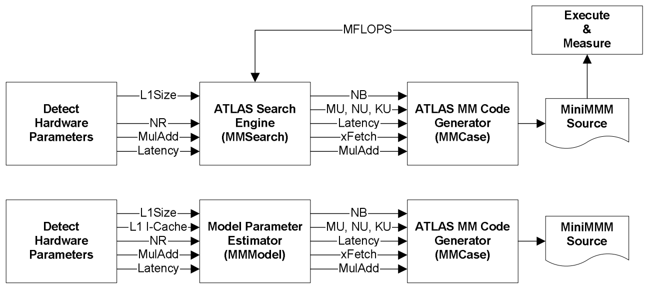

MFLOPS
Execute & Measure
Detect Hardware Parameters
L1Size
NR
MulAdd
Latency
ATLAS Search Engine (MMSearch)
NB
MU, NU, KU
Latency
xFetch
MulAdd
ATLAS MM Code Generator (MMCase)
MiniMMM Source
Detect Hardware Parameters
L1Size
L1 I-Cache
NR
MulAdd
Latency
Model Parameter Estimator (MMModel)
NB
MU, NU, KU
Latency
xFetch
MulAdd
ATLAS MM Code Generator (MMCase)
MiniMMM Source

Figure 1. Empirical and model-driven optimizers

This paper provides the first quantitative evaluation of the differences between empirical and conventional, model-driven optimization. Figure 1 shows our experimental set-up. Like all systems that use empirical optimization, ATLAS has (i) a module that performs empirical search to determine certain parameter values (MMSearch), and (ii) a module that generates code, given these values (MMCase). We first studied the code generation module, and determined that the code it produces can be viewed as the end result of applying standard compiler transformations to high-level BLAS code. We then built a modified version of ATLAS in which the search module was replaced with a module (MMModel) that uses architectural models to estimate values for the same parameters that ATLAS normally searches for. Finally, we measured the performance of the code generated by the two systems on three architectures, and studied the generated code itself to understand performance differences. Since both ATLAS and our modified version use the same code generator, we are assured that any differences in performance arise solely from differences between empirical optimization and model-driven optimization as implemented in the two systems, and not from different code generation strategies.

This paper summarizes our findings. In Section 2, we use the framework of restructuring compilers to describe the code generation strategy of ATLAS. In Section 3, we describe how ATLAS determines the values of the optimization parameters by using an extensive empirical search. In Section 4, we describe novel program and machine models that we use to estimate values for these parameters without doing any empirical searches. In Section 5, we contrast the two approaches, comparing

- the time spent to determine the parameter values,
- the values of the parameters, and
- the relative performance of generated code.

In Section 6, we show how performance changes as parameter values are changed. This sensitivity analysis is useful for determining which parameters must be estimated most accurately for efficient code. We conclude in Section 7 with a discussion of our main findings, and suggest future directions of research.

2. HIGH-PERFORMANCE BLAS

In this section, we use the framework of restructuring compilers to describe how ATLAS produces highly optimized code for the Basic Linear Algebra Subroutines (BLAS). This description is based solely on what we have been able to deduce by studying the source code of the ATLAS system. The code produced by ATLAS depends on certain optimization parameters, which are assumed to be given to the code generator by an oracle for now.

2.1 High-level BLAS code

We restrict our attention to the Level-3 BLAS, which are by far the most complex and time-consuming of the BLAS routines. The simplest Level-3 BLAS performs the following computation:

$$C = \alpha A \times B + \beta C \qquad \textbf{(1)}$$

In this equation, A, B and C are input matrices of appropriate shape, while α and β are scalars. Note that matrix multiplication is a special case of this computation in which $\alpha = 1$ and $\beta = 0$. We will consider the case $\alpha = \beta = 1$, for which it is straightforward to provide an implementation in a high-level, C-like language, as shown in Figure 2.

Other BLAS-3 routines perform the same computation with transposed versions of either *A* or *B* or both. Level-2 BLAS codes perform variations of matrix-vector multiplication, while Level-1 BLAS codes perform vector operations such as inner product and sum.

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

Figure 2. Matrix Multiplication

It is well known that the code in Figure 2 will perform poorly if the matrices *A*, *B*, and *C* are large. This is because modern computers have deep processor pipelines and multi-level memory hierarchies consisting of caches and registers. In principle, matrix multiplication has excellent algorithmic data reuse because it performs $O(N^3)$ operations on $O(N^2)$ data. In practice, the program may run poorly if a lot of data is touched between successive accesses to a given cache line, evicting the cache line before it can be accessed again. For example, in the code shown in Figure 2, successive accesses to a given element of B are separated by accesses to $O(N^2)$ data, so every access to B may miss in the cache. Registers may be considered to be the highest level of the memory hierarchy. To use them effectively in matrix multiplication, it is necessary to register-allocate array elements, which many compilers do not normally do. Finally, on pipelined processors that issue instructions in order, there will be little overlap in the execution of different iterations of the k-loop, so instruction-level parallelism (ILP) will not be exploited.

2.2 Optimized BLAS codes

To address these performance problems, it is necessary to restructure the code to exploit features of modern architectures. ATLAS is not a restructuring compiler, but the code it produces can be viewed as the end result of performing the following sequence of code transformations on the high-level code in Figure 2.

- *Cache-level tiling*: The standard approach to exploiting data reuse in loops such as matrix multiplication is to tile (or block) the loops in the loop nest [1]. In effect, the matrix multiplication is converted to a sequence of smaller matrix multiplications whose working sets fit in the cache. Each of the small matrix multiplications multiplies an *MB*x*KB* sub-matrix of A by a *KB*x*NB* sub-matrix of B and accumulates the result in a *MB*x*NB* sub-matrix of *C*. In this paper, we call these operations *mini-MMMs*. ATLAS tiles only for the L1 data cache, and it uses only square tiles (*NB=MB=KB).* The code for a mini-MMM is shown in Figure 3, assuming the JIK loop order.

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

Figure 3. Mini-MMM code

The value of *NB* is an optimization parameter. Choosing too large or too small a value of NB increases the L1 cache miss ratio and leads to inefficient cache utilization.

- *Register-level tiling*: The code for the mini-MMM in the previous step is itself tiled to make effective use of the general-purpose registers. Each of the smaller matrix multiplications multiplies a *MU*x*1* sub-matrix of A with a *1*x*NU* sub-matrix of B and accumulates the result in a *MU*x*NU* sub-matrix of *C*. In this paper, we call these *micro-MMMs*. The loops of a micro-MMM are unrolled completely to produce a straight-line code segment.

The pseudo-code for a mini-MMM after register-level tiling and unrolling of the micro-MMM is shown in Figure 4. In this code, we assume for simplicity that the elements in the A, B, and C tiles touched in the mini-MMM are indexed starting at (0, 0). A pictorial view of this code is shown in Figure 5. The shaded regions in this figure correspond to a micro-MMM.

```
for (int j = 0; j < NB; j += NU)
  for (int i = 0; i < NB; i += MU)
    load C[i..i+MU-1, j..j+NU-1] into registers
    for (int k = 0; k < NB; k++)
       load A[i..i+MU-1,k] into registers
       load B[k,j..j+NU-1] into registers
       multiply A's and B's and add to C's
    store C[i..i+MU-1, j..j+NU-1]
```

Figure 4. Mini-MMM Code after Register-level Tiling

After register-level tiling, the k loop in Figure 4 is unrolled completely if *NB* is small enough. Otherwise, this loop is unrolled by a factor of *KU*. Unrolling together with scheduling of operations, as described in the next step, gives the effect of software-pipelining the innermost loop of the mini-MMM.

The values of *MU*, *NU*, and *KU* are optimization parameters. If *MU* and *NU* are too small, registers are not fully utilized, but if they are too large, the compiler may generate many spills to memory. Unrolling by *KU* reduces loop control overhead, but too much unrolling can lead to instruction cache overflow, which reduces performance.

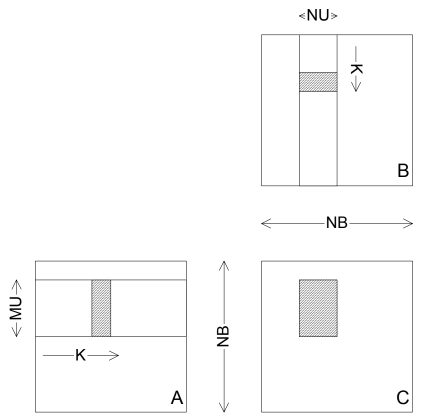


Figure 5: Pictorial View of Code in Figure 4

- *Scheduling:* The body of the innermost loop that results from the previous step is straight-line code that contains *KU* copies of the micro-MMM code; each copy has *MU*NU* multiply-add operations and the corresponding *MU* loads from A and *NU* loads from B. The operations in the loop body can be divided into two groups: *computation* and *memory accesses*.

We first focus on the scheduling of the computation operations in the loop body, assuming that the appropriate loads have been done. If the MMSearch routine in ATLAS detects that the processor has a combined multiply-add instruction (that is, the parameter *MulAdd* is true), the code generation module generates code that increases the likelihood that the C compiler will uses that instruction. We confine our discussion to the more complicated case when such an instruction is not available. In this case, each multiply instruction writes its result to a temporary register, which is read by the corresponding add instruction.

For efficient pipelining, it is desirable that a multiplication instruction and its corresponding addition instruction be separated by independent instructions. To accomplish this, instruction scheduling considers separately the *MU*NU* multiply instructions ($M_1M_2\ldots M_{MU*NU}$) and the *MU*NU* add instructions ($A_1A_2\ldots A_{MU*NU}$) in a single micro-MMM. It interleaves the two lists after skewing them by *Latency*, a parameter related to FP multiplier latency, to produce a schedule like this:

M_1
M_2
…
$M_{Latency}$
A_1
$M_{Latency+1}$
A_2
$M_{Latency+2}$
…
$M_{MU*NU-1}$
$A_{MU*NU-Latency}$
M_{MU*NU}
$A_{MU*NU-Latency+1}$
$A_{MU*NU-Latency+2}$
…
A_{MU*NU}

Such a schedule requires *Latency* number of extra registers to hold the results of the multiplications. In particular, if *NR* is the number of registers, the following inequality must hold for efficiency:

$$MU * NU + MU + NU + Latency \leq NR \qquad \textbf{(2)}$$

It is obvious that the final *Latency* adds can overlap with the initial *Latency* multiplies of the next iteration of the K loop.

We now consider where the loads of A and B elements must be inserted into this computation schedule. A naïve approach is to schedule all the loads as early in this schedule as dependences allow. However, if the CPU supports only a small number of outstanding loads, the instruction pipeline might stall. To avoid this, ATLAS schedules blocks of *NFetch* loads at a time, where *NFetch* is an optimization parameter that depends on the number of outstanding loads that the CPU supports. In this way, the executions of memory and ALU operations are interleaved.

Notice that in each micro-MMM, there are *MU+NU* loads but *MU*NU* multiplies and adds. Therefore, the last part of this schedule for the loop body is likely to have only arithmetic operations. To make better use of memory, ATLAS tries to overlap some of the initial loads for one iteration of the k loop with the final computations of the previous iteration. This step can be viewed as a simple kind of software pipelining of the unrolled loop. The number of these initial loads is determined by an optimization parameter called *IFetch*.

Finally, there is a parameter called *FFetch* whose precise role is somewhat unclear to us. It appears to be used to suppress the initial loads for C from memory under some circumstances.

The values of *MulAdd*, *FFetch*, *IFetch*, *NFetch*, and *Latency* are optimization parameters.

2.3 Versioning

As in many BLAS libraries, the library generated by ATLAS actually contains several versions of each high-level BLAS-3 algorithm. For example, it is often the case that A, B, and C are submatrices of larger matrices. In that case, it may be beneficial to copy these matrices into contiguous storage to reduce conflict misses during the execution of the operation. However, the overhead of copying may not be worthwhile if the matrix sizes are small, or if copying requires more memory than what is available. Most BLAS libraries therefore have both a copying and a non-copying version of each code; at runtime, matrix size information is used to determine which version should be executed. Decisions that ATLAS makes at runtime include *(i)* whether to copy the tiles for mini-MMM into continuous memory, *(ii)* the loop order of the mini-MMM (only JIK or IJK are considered), and *(iii)* how boundary sub-matrices will be multiplied. Boundary sub-matrices arise because a matrix dimension may not necessarily be an integer multiple of the tile size. Skinny matrices “left-over” from tiling can be multiplied by specialized code called *clean-up code*, in which the exact size of operand sub-matrices is used to fully unroll loops. At runtime, ATLAS decides whether to call the generic MMM code or the clean-up code for handling the boundary sub-matrices.

2.4 Summary

The code produced by ATLAS can be viewed as the end-result of applying a sequence of well-known program transformations to high-level BLAS codes. The optimization parameters used in these transformations are *NB*, *MU*, *NU*, *KU*, *MulAdd*, *Latency*, *IFetch*, *NFetch*, and *FFetch.*

Although these optimization parameters are specific to ATLAS, we would argue that most of them would arise naturally in the context of conventional restructuring compilers. Any compiler that tiles for the data cache will compute a parameter similar to *NB*. Parameters similar to *MU*, *NU*, and *KU* are used for register tiling. Instruction selection and scheduling would require parameters like *MulAdd*, *Latency* and something similar to *NFetch* or *IFetch*. *FFetch* is the odd man out.

3. HOW ATLAS FINDS PARAMETER VALUES

As mentioned earlier, ATLAS does its work in two phases.

- The installation phase of ATLAS is shown in Figure 1. First, ATLAS computes machine parameters such as L1 data cache size and the number of registers. Then, it performs empirical search to determine values for the optimization parameters, using machine parameters to limit the size of the search space. Finally, it generates all the mini-MMM versions for the library.
- At run-time, an application program calls ATLAS’s general interface routine. This interface routine takes care of some trivial cases such as empty input matrices and the case when $\alpha = 0$ (which means the result is just βC), and then makes a sequence of calls to the appropriate mini-MMM to perform the matrix multiplication.

3.1 Estimating machine parameters

Since ATLAS is self-tuning, it does not require the user to provide the values of machine parameters. Instead, it runs micro-benchmarks to determine approximate values for most of these parameters. Among these are

- the size of L1 data cache (*L1Size*),
- the number of floating-point registers (*NR*),
- the availability of a multiply-add instruction (*MulAdd*), and
- the latency of the floating-point multiply unit (*Latency*).

These micro-benchmarks have nothing to do with matrix multiplication; for example, the micro-benchmark for estimating the size of the L1 data cache is similar to the one in Hennessy and Patterson [9].

As described in Section 2, two other architectural parameters are critical for performance: *(i)* the L1 instruction cache size, and *(ii)* the number of outstanding loads that the machine supports. ATLAS does not determine these explicitly; instead, they are considered implicitly during the optimization of matrix multiplication code. For example, the size of the L1 instruction cache may limit the amount of unrolling of the k loop of Figure 4 (parameter *KU*) that is beneficial. Rather than estimating the size of the instruction cache by running a micro-benchmark and then using that to determine the amount of unrolling, ATLAS generates a suite of MMM kernels with different amounts of unrolling, and determines the best one experimentally.

3.2 Estimating optimization parameters

Once machine parameters have been estimated, ATLAS estimates optimization parameters using an extensive search to find optimal values.

The optimization sequence is as follows.

1. Find best *NB*.
2. Find best *MU* and *NU*.
3. Find best *KU*.
4. Find best *Latency*.
5. Find best *Fetch* factors.
6. Find non-copy version crossover.
7. Find optimal cleanup codes.

We now discuss each of these steps in greater detail.

3.2.1 Find Best NB

In this step, ATLAS generates a number of mini-MMMs for matrix sizes *NB*x*NB* where *NB* is a multiple of 4 that satisfies the following inequalities:

$$16 \le NB \le 80; NB^2 \le L1Size \quad \textbf{(3)}$$

A "phase-ordering problem" in generating these mini-MMM codes is that ATLAS does not as yet have optimal values for the other optimization parameters. Therefore, it uses rough estimates for the values of these parameters. The values of *MU* and *NU* are set to the values closest to each other that satisfy (2). For each matrix size, ATLAS tries two extreme cases for *KU* – no unrolling (*KU*=1) and full unrolling (*KU=NB*). Suitable *Latency* and all *Fetch* parameters are obtained from running the micro-benchmarks.

The *NB* that produces highest MFLOPS is chosen as "best *NB*" value, and it is used from this point on in all experiments as well as in the final versions of the optimized mini-MMM code.

3.2.2 Find Best MU and NU

This step is a straightforward search that refines the reference values of *MU* and *NU* that were used to find the "best *NB*". ATLAS tries all possible combinations of *MU* and *NU* that satisfy Inequality (2). The cases when *MU* or *NU* is 1 are treated specially. A test is performed to see if 1x9 unrolling or 9x1 unrolling is better than 3x3 unrolling. If not, unrolling factors of the form 1x*U* and *U*x1 for values of *U* greater than 3 are not checked.

3.2.3 Find Best KU

This step is another simple search. Unlike *MU* and *NU*, *KU* does not depend on the number of available registers, so technically we can make it as large as we want to without causing register spills. The main constraint here is instruction cache size. ATLAS tries values for *KU* between 4 and *NB*/2 as well as the special values 1 and *NB*. The value that gives best performance in terms of MFLOPS (based on *NB*, *MU* and *NU* as determined from the previous steps) is declared the optimal value for *KU*.

3.2.4 Find Best Latency

In this step, ATLAS tries different schedules for the computations in the unrolled k loop of Figure 4 to determine if there is a skew that generates a better schedule than the one obtained by using the hardware *Latency* value. It checks all the values between 1 and 6, and selects the one that performs best, using parameter values determined from the previous steps. It also ensures that the chosen value divides *MU*NU*KU* to facilitate instruction scheduling.

3.2.5 Find Best Fetch

In this step ATLAS searches for the values of *FFetch*, *IFetch* and *NFetch*. First, ATLAS determines the value of FFetch (0 or 1). Then, it searches for the best value of the pair (*IFetch*, *NFetch*) where *IFetch* is in the interval [2,*MU*+*NU*] and *NFetch* is in the interval [1,*MU*+*NU*-*IFetch*].

3.2.6 Find Non-Copy Version of NB

ATLAS generates both a copy and a non-copy version of the mini-MMM code. Without copying, the sub-matrices touched by a mini-MMM are not stored contiguously in memory, so the probability of conflict misses is higher than in the copy version. To avoid excessive conflict misses, ATLAS uses a smaller tile size for the non-copy version than for the copy version. It searches for tile sizes from *NB* down to 16 in steps of 4, until performance deteriorates by 20% or more. The tile size that yields highest performance is selected to be the value of *NCNB* (Non-Copy *NB*).

ATLAS also does a very restricted search for unroll factors and latencies (between 2 and 9 in this case). These searches are very much along the lines of the corresponding copy versions.

At runtime, ATLAS invokes the copy version whenever the collective size of the matrices is big enough that the cost of copying can be amortized by computation. The non-copy version is used when the dimensions of the matrices (see Figure 2) satisfy the following constraint:

$$M * N * K \le NB^3 \quad \textbf{(4)}$$

One other case in which ATLAS decides to use the generated non-copy version is for matrices larger than (2^{18}-2)/*NB* in one dimension.

3.3 Generate Optimal Cleanup Codes

If the tile size is not a multiple of the original matrix size, there may be "left-over" rows and columns in the matrices to be multiplied that are too few to form a tile. ATLAS generates "clean-up" code for handling these left-over rows and columns. These special tiles have one dimension of size *NB* and another dimension between 1 and *NB*-1. The size of this other dimension will be called *L* in this discussion.

ATLAS generates cleanup codes as follows. For each value of *L* from *NB*-1 on down, ATLAS generates a specialized version of the code in which some of the loops are fully unrolled. Full unrolling is possible because the shapes of the operands are completely known. When the performance of the general version falls within 1% of the performance of the current specialized version, the generation process is terminated. The current *L* is declared to be the *Crossover Point.* At runtime, the specialized versions are invoked when the dimension of the left-over tile is greater than *L*, while the general version is invoked for tile sizes smaller than *L*.

4. ESTIMATING PARAMETERS USING MODELS

We now discuss the use of architectural models to estimate the values of optimization parameters without empirical search.

4.1 Estimating *NB*

There is a large body of work in the compiler community on estimating good tile sizes in the context of general-purpose compilers. Dongarra and Schreiber [6] determine tile sizes and orientations such that the amount of data touched by the tile is bounded by the size of the cache, while minimizing the surface to volume ratio of the tile. Their solution modeled the problem as a constrained minimization problem. Boulet *et al* [2] suggest that the surface to volume ratio is not the appropriate metric to be minimized and present alternative metrics one might choose to optimize. Neither of these two approaches addresses the question of conflict misses. Lam, Rothberg and Wolf present strategies to determine square tiles while minimizing capacity and conflict misses [12]. Coleman and McKinley generalized this analysis to rectangular tiles [5]. Ramanujam and Sadayappan have considered tiling in the context of distributed-memory computers. Wolf, Chen and Maydan [20] discuss how tile sizes can be determined in the setting of production compilers. Clauss has used Ehrhart polynomials to provide exact values for the working set of a loop [4]. Unfortunately, few if any of these papers have comparisons with hand-optimized BLAS code, so it is difficult to determine the accuracy of these methods and their impact on performance.

Our model for estimating *NB* is quite intricate, so we describe it in stages. Although the model works for any level of the memory hierarchy, we consider only the L1 cache (like ATLAS).

First, we assume a simple cache model:

- fully-associative cache (no conflict misses);
- line size is one element (no spatial locality);
- optimal replacement strategy (not LRU).

Consider the code for matrix multiplication in Figure 2 in which the loops are executed in the JIK order. We can distinguish between three different scenarios, depending on how large the matrices are compared to the cache size.

- Large Cache Model (LCM): This model is valid when the size of the matrices is small compared to the cache size. In this model, the only misses are cold misses because nothing ever needs to be evicted from the cache.
- Small Cache Model (SCM): This model is valid when the cache is small compared to the size of the matrices; intuitively, it is too small to hold even one row of a matrix. Specifically, if two different accesses to an array element are separated by accesses to *O(1)* other elements, we assume that the second access hits in the cache; otherwise, the number of accesses to other elements grows with matrix size, so we assume that the second access misses in the cache.
- Medium Cache Model (MCM): This model is valid when the cache is big enough to hold one (or several) rows of a matrix, but is not enough to hold a full matrix.

These models can be used to make quantitative performance predictions, but we will not pursue that line of investigation in this paper. Our objective here is only to choose the largest *NB* that makes the LCM valid during the execution of the mini-MMM code shown in Figure 3. To keep things simple in the discussion that follows, we will use the word matrix instead of matrix tile.

Matrix A is indexed by the control variables of the innermost two loops, *i* and *k*. Therefore, every iteration of the outermost loop *j* touches all of A. To stay in LCM, all misses must be cold misses, so we need to be able to store A completely in the cache. This will require storage for $MB*KB=NB^2$ elements.

Matrix B on the other hand is accessed by the control variables of the outermost loop *j* and the innermost loop *k*. Therefore once *j* is fixed, we will need to access the entire j^{th} column of B in every iteration of loop *i*. After this computation, we will not access this column again, so we need storage for *KB=NB* elements of B in the cache.

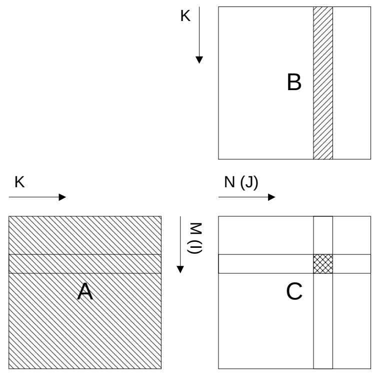


Figure 6. Matrix Indexing Scheme and Cache Usage for JIK

Finally, matrix C is indexed by the control variables of the outermost two loops – *j* and *i*. This means that we fix a single element of C, reuse it in all the iterations of the innermost loop *k*, and never touch it again after that. Therefore we need storage for one element of C in the cache.

Summarizing, we need $NB^2 + NB + 1$ lines in the cache to satisfy the requirements for LCM in this simple cache model. Because we know the capacity C of the cache, we can set *NB* to be the largest value consistent with this inequality:

$$NB^2 + NB + 1 \le C \tag{5}$$

For loop orders other than JIK, the reasoning is very similar. The only difference is that the matrices A, B and C change their roles, thus contributing different terms on the left side of (5), but the final inequality is the same. In summary, we always need to keep one full matrix in the cache, a row or a column of another matrix and a single element of the third matrix.

4.1.1 *Modeling NB for Caches with Larger Lines*

We now refine our cache model by allowing lines to hold some number of elements greater than 1 (say *B*). Spatial locality becomes important, and we must consider the layout of matrices in the storage space.

If we now return to our example (loop order JIK), and take into account that we are dealing with FORTRAN matrices stored in column-major layout, we can correct Inequality (5) as follows:

$$\left\lceil \frac{NB^2}{B} \right\rceil + \left\lceil \frac{NB}{B} \right\rceil + 1 \le \frac{C}{B} \tag{6}$$

The reasoning behind Inequality (6) is that we still need to cache the full matrix A, which contains $MB*KB=NB^2$ elements. With *B* elements per line and assuming that A is laid out sequentially in memory, we need $\lceil NB^2 / B \rceil$ cache lines. We need also to cache a full column of B (*KB=NB* elements). Because matrices are laid out in column major order in memory, one such column will require $\lceil NB/B \rceil$ cache lines. Finally the single element of C that we need to store in cache is part of a single cache line. The same reasoning also works for loop orders JKI and KJI.

For other loop orders (IJK, IKJ, KIJ) we need to store in the cache a full matrix, a matrix row from another matrix, and a single element from the third matrix (e.g. these matrices in the IJK case are B, A and C respectively). Since we have non-unit line sizes, storing a row in the cache requires more storage than a column does. Because of the column major layout, each element of the row will most likely be part of a different cache line, which makes the row require *NB* cache lines instead of the $\lceil NB/B \rceil$, normally required by a column. Therefore Inequality (6) changes to the following:

$$\left\lceil \frac{NB^2}{B} \right\rceil + NB + 1 \le \frac{C}{B} \qquad \textbf{(7)}$$

4.1.2 Modeling NB for LRU Replacement Caches

Finally, we analyze the effect of cache replacement policy. Most caches implement (pseudo) Least Recently Used (LRU) replacement policy. As we shall see, this has a substantial impact on the optimal NB value.

To find the impact of replacement policy, we must reason about the history of data accesses. A well-known approach for doing this is the *stack algorithm* [13]. In this approach, the history of memory references is represented as a stack that is updated after every memory reference M by pushing the line containing M onto the top of the stack and removing it from its old location if the line was already in the stack. An LRU cache would obviously keep only the top L lines, where L is the size of the cache.

We can rewrite the pseudo-code of the JIK loop order in the following way:

```
for j^th column B ($B_{Xj}$)
  for i^it row of A ($A_{iX}$)
    multiply $A_{iX}$ by $B_{Xj}$ and add to $C_{ij}$
```

The top of the stack after one execution of the inner multiply will look like:

$$A_{i,1}B_{1,j}A_{i,2}B_{2,j}\cdots A_{i,NB}B_{NB,j}C_{i,j} \qquad \textbf{(8)}$$

$C_{i,j}$ is used repeatedly (for each element multiply of A_{iX} and B_{Xj}), so it stays at the top of the stack.

This innermost multiply is performed *NB* times using the different rows of A and the same column of B. So the top of the stack after one full execution of the middle loop looks like this:

$$\begin{array}{l} A_{1,1}A_{1,2}\cdots A_{1,NB}C_{1,j} \\ \to A_{2,1}A_{2,2}\cdots A_{2,NB}C_{2,j} \\ \vdots \\ \to A_{NB-1,1}A_{NB-1,2}\cdots A_{NB-1,NB}C_{NB-1,j} \\ \to A_{NB,1}B_{1,j}A_{NB,2}B_{2,j}\cdots A_{NB,NB}B_{NB,j}C_{NB,j} \end{array} \qquad \textbf{(9)}$$

Notice that the elements of the j^{th} column of B appear only in the last *row* of the history because they are accessed in each iteration of the *i* loop.

At the next iteration of the outer loop, we need to access $A_{1,1}$ again. Since $A_{1,1}$ is the oldest reference in the history and the cache replacement policy is LRU, the line containing $A_{1,1}$ will be in the cache only if everything in the history so far is in the cache. The history contains matrix A (*NB*NB* elements), a column of B and a column of C. In contrast, in a cache with optimal replacement, we need only one element of C.

Continuing this argument, we see that we need space for two columns of B in the cache. With the space for the second column of B, it becomes possible for the elements of A to remain in the cache while the elements of the *j*th column of B are evicted.

Applying the same argument again, but this time for the columns of C, it is easy to show that we need storage for one extra element of C to ensure that no elements of A are evicted as a consequence of LRU replacement.

Putting all this together, we get the following inequality:

$$\left\lceil \frac{NB^2}{B} \right\rceil + 2\left\lceil \frac{NB}{B} \right\rceil + \left(\left\lceil \frac{NB}{B} \right\rceil + 1 \right) \le \frac{C}{B} \text{ or}$$

$$\left\lceil \frac{NB^2}{B} \right\rceil + 3\left\lceil \frac{NB}{B} \right\rceil + 1 \le \frac{C}{B} \qquad \textbf{(10)}$$

In summary, for an LRU cache and loop order JIK, the optimal *NB* ensures that the full matrix A, two columns of B, and one column and one element of C fit in the cache. In general, for any other loop order, the optimal *NB* ensures that one full matrix, two columns or rows of another matrix, and one column or row and one element of the third matrix fit in the cache.

Table 1. Equations for Optimal *NB* in the General Case

Loop order	Equation
IJK, IKJ	$\left\lceil \frac{NB^2}{B} \right\rceil + 3NB + 1 \le \frac{C}{B}$
JIK, JKI	$\left\lceil \frac{NB^2}{B} \right\rceil + 3\left\lceil \frac{NB}{B} \right\rceil + 1 \le \frac{C}{B}$
KIJ	$\left\lceil \frac{NB^2}{B} \right\rceil + 2NB + \left(\left\lceil \frac{NB}{B} \right\rceil + 1 \right) \le \frac{C}{B}$
KJI	$\left\lceil \frac{NB^2}{B} \right\rceil + 2\left\lceil \frac{NB}{B} \right\rceil + (NB+1) \le \frac{C}{B}$

The inequalities for all the different loop orders are summarized in Table 1. The only variable in all these equations is *NB*, therefore we can find the largest integer solution for it using binary search.

4.1.3 Modeling NB for Set-Associative Caches

In practice, caches are not fully associative. In spite of this, the models developed so far are relevant for the copy version of mini-MMM code, which copies tiles into a sequential region of memory to reduce conflict misses.

To determine the optimal value for *NB* when copying is not performed, we need to estimate the impact of conflict misses. Our estimates are based on the model developed by Fraguela et al. [7], which uses a statistical approach to predict approximately the number of cache misses caused by a loop nest with constant bounds. For lack of space, we do not address this issue further in the paper.

4.2 Finding *MU*, *NU*, and *KU*

Register tiling can be looked at as a special case of tiling (or equivalently unroll and jam [1]) for a cache that has the following properties:

- fully-associative – any register can contain a value loaded from any memory address;
- unit line size – each register contains a single value and is a line in this L0 cache by itself;
- optimal replacement policy – the code generator is free to schedule any register fill or spill at any time.

We assume that the register tile has K as its outermost loop, as shown in Figure 4. If all three parameters (*MU*, *NU* and *KU*) are equal to some value *U*, we can use inequality (5) to constrain *U*:

$$U^2 + U + 1 \le NR \tag{11}$$

Here *NR* is the number of floating point registers. Intuitively, U^2 registers are required for the register tile of C, *U* registers are required for B and 1 register is required for A. Because *NR* is usually small, it may be sub-optimal to unroll equally in all directions. For example *NR*=6 forces us to make *U*=1 (no unrolling). However if we can unroll by different amounts in the three directions, we can choose to unroll by 2 in exactly one direction and get better performance.

If we think about the three unroll factors separately, (11) is replaced with:

$$MU * NU + NU + 1 \le NR \tag{12}$$

Notice that while *MU* and *NU* are constrained by the number of registers, unrolling along the K direction is not.

4.2.1 ATLAS-Compatible MMM Unroll Model

The code generation strategy described in Section 2.2 actually requires *MU* registers rather than just one register for A (this allows more elements of A to be prefetched into registers). Furthermore, in the absence of a combined multiply-add instruction we need extra registers for storing the result of floating-point multiplications. Taking into account all these considerations, the appropriate constraint is given by Inequality (2), where *Latency* is replaced by the number of temporary registers (*TR*) required for scheduling:

$$MU * NU + MU + NU + TR \le NR$$

Now if we assume *MU*=*NU* we get:

$$NU^2 + 2NU + (TR - NR) \le 0 \tag{13}$$

which we can solve for *NU*. Having obtained *NU*, we can solve (4) for *MU*:

$$MU = \frac{NR - TR - NU}{NU + 1} \tag{14}$$

and adjust *MU* and *NU* so that they are both at least one and *MU* is the bigger one.

For *KU*, our approach is to unroll along the K direction as much as possible without overflowing the L1 instruction cache. We can do this because we know the size of the instruction cache. When generating code for a specific *KU* value, we measure the size of the loop in bytes, using a special feature of the C language, present in the GCC compiler that allows us to compute addresses of goto-style labels. Here is an example:

```
printf(“code size = %d\n”, &&l2 - &&l1);
return;
l1:
  // mini-MMM code generated for fixed KU
l2:;
```

This code prints the number of bytes of generated binary code between the two labels.

4.3 Finding *Fetch*, *Latency*, and *MulAdd*

ATLAS has three fetch parameters: *FFetch*, *IFetch* and *NFetch*. We choose *FFetch*=1 (to prefetch the portion of C into registers). We believe that *IFetch* and *NFetch* should both be set to the number of supported outstanding loads (*OL*). However we do not yet have a benchmark that estimates *OL*; and we also found that performance was not sensitive to the values of these parameters, as we discuss in Section 6. Therefore we set both parameters to 2.

For the optimization parameters *Latency* and *MulAdd,* our model uses the machine parameters determined by the hardware parameter detection module.

4.4 Summary

We have developed a model-driven approach for choosing all optimization parameters used by ATLAS: *NB*, *MU*, *NU*, *KU*, *MulAdd*, *Latency*, *FFetch*, *IFetch* and *NFetch*.

5. PERFORMANCE ANALYSIS

In this section, we describe the results of our experiments with ATLAS and with the modified version of ATLAS (called Model from this point on) in which empirical optimization has been replaced with model-driven optimization. We compare both the installation time of the two versions, and the execution times of double-precision matrix multiplications of various sizes on the three architectures shown in Table 2.

5.1 The Installation Phase

Table 2. Test Platforms Hardware

	SGI	Sun	Intel
CPU	R12000	UltraSparcIII	PIII-Xeon
Frequency	270MHz	900MHz	550MHz
Registers	32	32	8
L1 Cache	32KB/32KB	64KB/32KB	16KB/16KB
L2 Cache	4MB	8MB	512KB
Memory	1GB	2GB	1GB
OS	IRIX64 v6.5	SunOS 5.8	RedHat 7.3
ATLAS Compiler	MIPSpro CC v7.3.1.1m	Forte 7 C v5.4	gcc v3.2
ATLAS Options	-O3 -64 -OPT:Olimit=15000 -TARG:platform=IP27 -LNO:blocking=OFF -LOPT:alias=typed	-dalign -fsingle -xO2 -native	-fomit-frame-pointer -O
Native Compiler	MIPSpro F77 v7.3.1.1m	Forte 7 F95 v7.0	g77 v3.2
Native Options	-O3 -64 -OPT:Olimit=15000 -TARG:platform=IP27	-dalign -native -xO5 –pad	-O3 -fno-inline -funroll-all-loops -funroll-loops

The installation phase can be divided into four parts. The first part, *Detecting Machine Parameters*, takes 6%-12% longer in Model, mainly because of the way the code is organized. The original version of ATLAS detects the cache size as part of the empirical search while Model performs this task in this part. The second part, *Estimating Optimization Parameters*, is where ATLAS performs the empirical search. In Model, this part takes almost no time because no search is performed. The final two parts of the installation phase generate the final source, and then compile it to make the library. Currently, we generate more versions of the clean-up code (for multiplying boundary sub-matrices) than ATLAS does, so there are minor differences between ATLAS and Model in the time they take to execute these two parts. Figure 7 presents this breakdown of installation times.

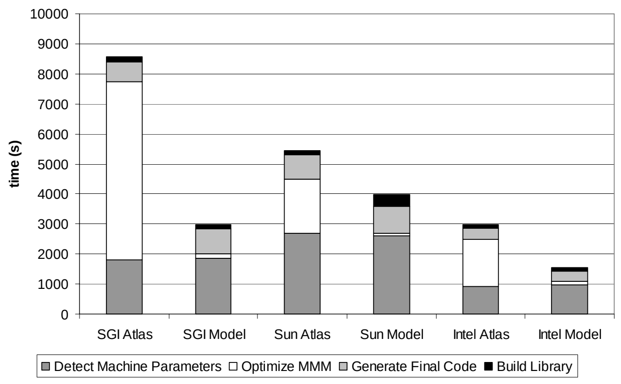


Figure 7. Installation Time

5.1.1 *Optimization Parameter Values*

Table 3 and Table 4 show the values of the optimization parameters that are determined by ATLAS and by Model respectively. The parameter values determined by the two systems are very similar on the Intel and SGI machines. On the Sun, tile size and KU values are significantly different, but other parameter values are close. The impact of these differences will be discussed later in this paper.

Table 3. ATLAS Estimated Parameters

Architecture	Tile Size	Unroll	Fetch	Latency
	Copy / Non-Copy	MU / NU / KU	F / I / N	
SGI	64 / 64	4 / 4 / 64	0 / 5 / 1	3
Sun	48 / 48	5 / 3 / 48	0 / 3 / 5	5
Intel	40 / 40	2 / 1 / 40	0 / 3 / 1	4

Table 4. Model Estimated Parameters

Architecture	Tile Size	Unroll	Fetch	Latency
	Copy / Non-Copy	MU / NU / KU	F / I / N	
SGI	62 / 45	4 / 4 / 62	1 / 2 / 2	6
Sun	88 / 78	4 / 4 / 88	1 / 2 / 2	4
Intel	42 / 39	2 / 1 / 42	1 / 2 / 2	3

5.2 Comparison of Performance

In this section, we compare the execution times of both the mini-MMM routines (Table 5) as well as complete MMM for various matrix sizes (Figure 8, Figure 9 and Figure 10).

5.2.1 *Mini-MMM Performance*

Table 5 shows that on both the SGI and Intel machines, the performance of the codes generated by the two approaches is similar. On the Sun, there is roughly a 20% difference in performance.

Table 5. Mini-MMM Performance Comparison

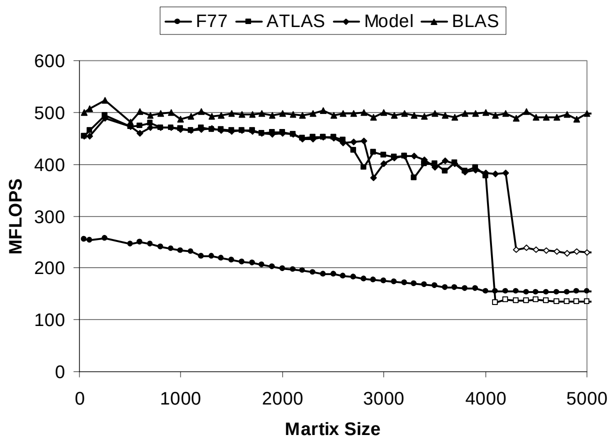
Architecture	ATLAS	Model	Difference
	(MFLOPS)	(MFLOPS)	(%)
SGI	457	453	1
Sun	1287	1052	20
Intel	394	384	1

5.2.2 *MMM Performance*

Next, we compare the performance of complete MMM using:

- mini-MMMs generated by ATLAS (with empirical search),
- mini-MMMs generated by Model,
- hand-tuned BLAS routines, and
- high-level matrix multiplication compiled using the most powerful optimizations available in the native compiler.

We compare performance for square matrices of size 100 to 5000. On SGI and Sun, both ATLAS and Model use the non-copy versions of mini-MMM for multiplying large matrices. These data-points are shown as *unfilled* markers on the plots.



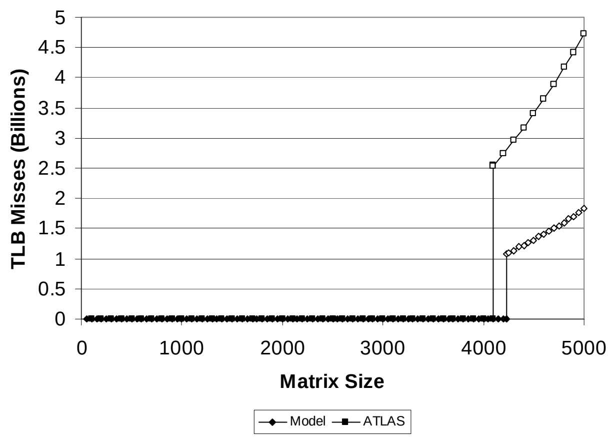


Figure 8. MMM Performance Comparison on SGI

On the SGI machine, the best performer is the native BLAS library. On the matrix sizes we tested, Model is always within 2% of ATLAS in performance. For matrix sizes larger than 4000, Model outperforms ATLAS by roughly 30%, but both are much slower than BLAS. For these matrix sizes, both ATLAS and Model decide to use the non-copy version, and this causes TLB misses to go up, as can be seen in Figure 8. ATLAS finds a tile size of 64, which is also the size of the TLB on the SGI machine. The model predicts a tile size of 45, so it requires fewer TLB entries, and thus performs better. These experiments demonstrate the well-known fact that for large data sizes, TLB effects can be important.

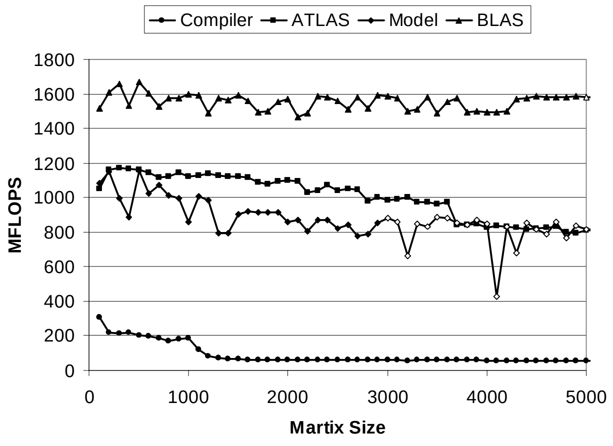


Figure 9. MMM Performance Comparison on Sun

On the Sun, the best performer is again the native BLAS library. The codes generated by ATLAS and by Model are between 25% and 50% slower than the BLAS. ATLAS-generated code performs about 20% better than Model-generated code for matrix sizes less than 3000.

As on the other machines, the native BLAS library performs best on the Pentium. Both ATLAS-generated code and Model-generated code perform about 20% worse than the BLAS, and are within 3% to 10% of each other.

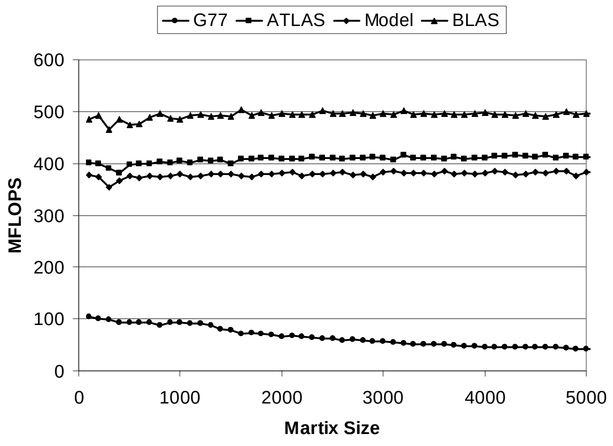


Figure 10. MMM Performance Comparison on Intel

The native compilers on all three machines did not produce very good code. In these experiments, the sizes of the matrices were parameters to the high-level code for matrix multiplication that was given to these compilers. We found that if the matrix sizes are hard-coded constants in this code, the performance obtained by the native compilers on SGI and Sun is close to that of ATLAS and Model. We do not yet understand this issue.

5.3 Summary

Two surprising conclusions can be drawn from the experimental results in this section. First, we found that handwritten BLAS libraries perform better than either ATLAS-generated or Model-generated code on all three machines; on the Sun and Intel machines, the difference in performance is 25%-33%. This suggests there is considerable room for improvement in both empirical and model-driven optimization techniques for generating the BLAS. Second, we found that on the SGI and Intel machines, the code generated by model-driven optimization is similar in performance to the code generated by ATLAS. On the Sun, only the values selected for tile size and *KU* by the two systems were significantly different, and the performance of Model-generated code is about 20% worse than ATLAS-generated code. It would appear that for generating optimized BLAS, empirical search is not as important as is commonly believed.

We also repeated these experiments with *rectangular* matrices of different sizes, but reached the same conclusions.

6. SENSITIVITY ANALYSIS

The results of the previous section show that the performance of code produced by model-driven optimization can be comparable to that of code generated by empirical optimization. An interesting question at this point is the following:

How sensitive is the performance of the code to changes in the values of optimization parameters?

This question is of interest for several reasons. In our context, the problem of generating efficient code can be viewed as a multi-dimensional optimization problem in which the independent variables are the optimization parameters such as *NB*, *MU*, *NU*, etc., and the dependent variable is performance. When the parameter values determined by the two systems are different, sensitivity analysis is useful to understand which of these differences affected performance the most. In addition, if it turns out that near the optimal point, performance is relatively insensitive to changes in one of the parameters, we can spend less time and effort in optimizing the value of that parameter, which would benefit both empirical and model-driven optimization. Such insights would also help in developing hybrid optimization strategies that combine model-driven and empirical optimization; if performance is insensitive to the value of some parameter, we can use simple models to choose its value, and use complex models or empirical optimization only for determining values for high-sensitivity parameters. In the limit, if performance near the optimal point is relatively insensitive to changes in *any* of the parameters, a simple model-driven optimization strategy is adequate.

Because of the large number of optimization parameters, it is impractical to vary all of them simultaneously. Instead, we set all optimization parameters to the values found by ATLAS, and then measured how performance of the mini-MMM code changes when we vary one parameter at a time.

On some graphs presented in this section, we mark three important points: "A" shows the parameter value selected by ATLAS, "M" shows the parameter value selected by the Model, and "B" shows the best parameter choice among those tested. *It is important to note that "M" does not represent the performance achieved by the model since all parameters other than the one being varied are set to ATLAS-selected values.*

6.1 Sensitivity to tile size (*NB*)

Figure 11, Figure 12, and Figure 13 show how performance changes when tile size is varied on the SGI, Sun and Intel machines respectively.

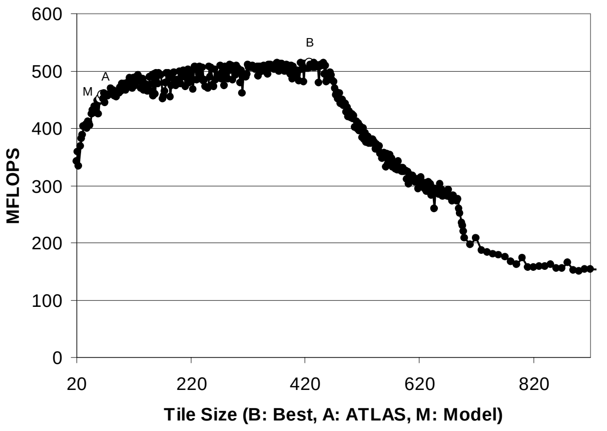


Figure 11: Performance vs. tile size on SGI

On the SGI machine, ATLAS and our model choose almost the same tile size (64 vs. 62). Notice that the best performance is obtained when the tile size is roughly 450; at that point, performance is roughly 525 MFLOPS, which is about 15% better than the performance obtained by ATLAS or the model. Since ATLAS uses the size of the L1 cache to limit the search space for *NB*, it does not explore such large tile sizes.

Our conjecture is that this large tile size is appropriate for the L2 cache. If we use our model to determine *NB* for the L2 cache, we obtain 722, which is close to the point at which there is a sudden drop in performance in Figure 11. Multi-level caching is outside the scope of this paper because we use the ATLAS infrastructure for code generation. We are investigating this matter further.

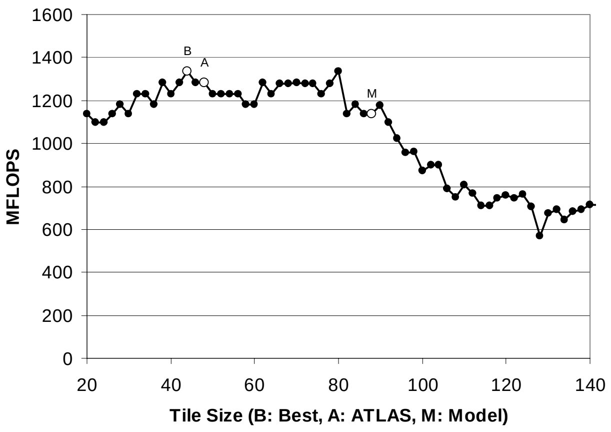


Figure 12: Performance vs. tile size on Sun

On the Sun machine, performance is relatively unchanged for tile sizes between 40 and 80. The best performance of 1336 MFLOPS is obtained for a tile size of 48. The model predicts a somewhat larger tile size (88), and this accounts for roughly 10% of the 20% difference in the performance of Model-generated and ATLAS-generated code. We are doing more detailed cache simulations to understand this issue.

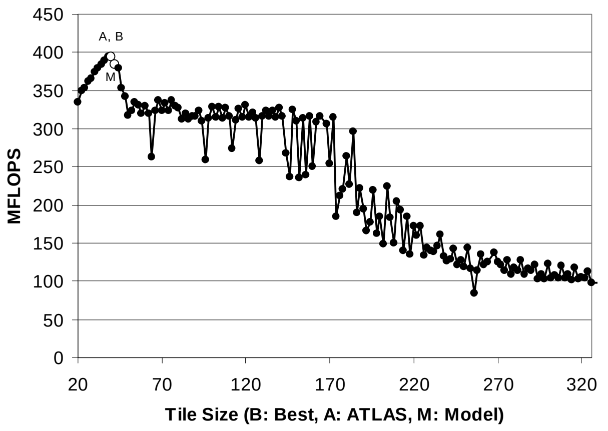


Figure 13: Performance vs. tile size on Intel

Sensitivity to tile size is most pronounced on the Intel machine; a 10% change in the tile size from the optimal tile size can reduce performance by 10% (400 MFLOPS down to 350 MFLOPS). However, both ATLAS and Model choose good tile sizes.

We believe that the sensitivity of performance to tile size on the Intel machine arises from the relatively small L1 cache size. In contrast, performance on the SGI and Sun machines is relatively insensitive to tile size.

6.2 Sensitivity to register tile size (*MU, NU*)

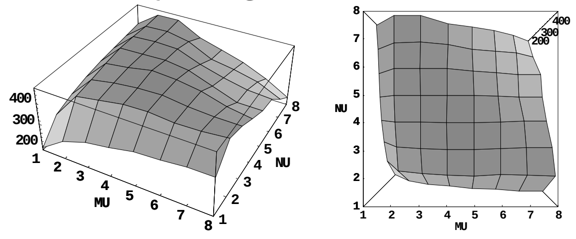


Figure 14: Performance vs. register tile size on SGI

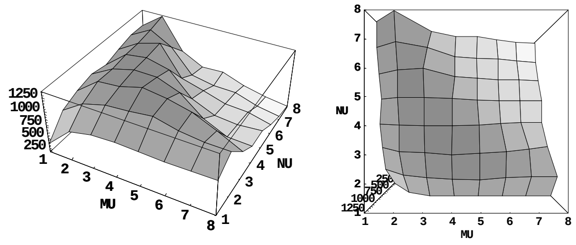


Figure 15: Performance vs. register tile size on Sun

Figure 14, Figure 15, and Figure 16 show how performance changes when the register tile size is changed on the SGI, Sun, and Intel machines respectively. Both ATLAS and Model choose the best register tiles on the SGI and the Intel machines. On the

SUN, ATLAS and Model choose slightly different register tiles, but both perform well.

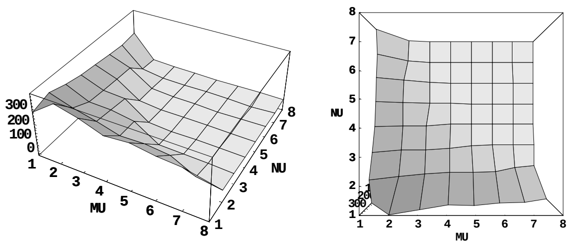


Figure 16: Performance vs. register tile size on Intel

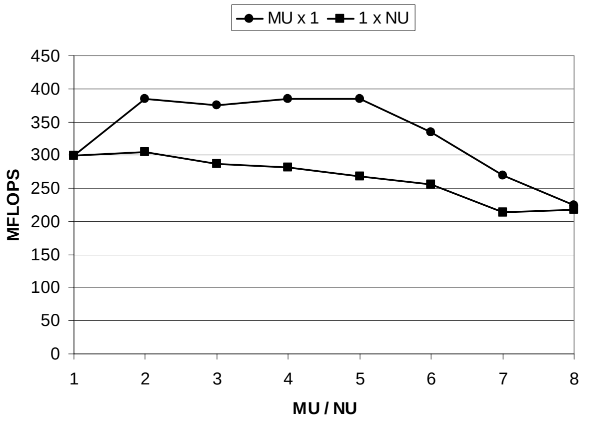


Figure 17: Performance vs. Register Tile Shape on Intel

Figure 17 shows that the *shape* of the register tile can affect performance significantly. For example, the value of the expression (*MU***NU*+*MU*+*NU*) is the same for (*MU*, *NU*) = (3, 1) and for (*MU*, *NU*) = (1, 3), but it can be seen that the performance of the code produced in the two cases differs by 33% on the Pentium.

The dependence of performance on the shape of the register tile disappears when we use icc rather than gcc to generate code. It appears that icc uses RISC-like instructions such as *load reg1,@mem* followed by *fmul reg1,reg2* instead of CISC-like instructions such as *fmul reg,@mem.*

6.3 Sensitivity to *KU*

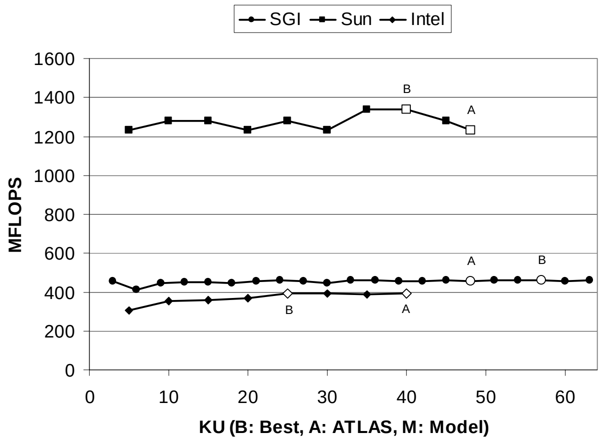


Figure 18: Performance vs. ***KU***

Figure 18 shows that performance of the code increases slightly as the unroll factor *KU* is increased. The performance points for Model lie outside the range shown, so they do not appear in this graph. Although ATLAS and Model choose very different values for *KU* on the Sun, it can be seen that performance is insensitive to this difference. The instruction caches on all the machines are large enough that the k loop in the mini-MMM code of Figure 4 can be unrolled completely without overflowing the cache. This will not be the case on a machine with a small instruction cache but a large data cache, because *NB* for such a machine would be too large to permit the loop to be unrolled completely. Similarly, full unrolling can be suboptimal on a machine with many registers, since the values of *MU* and *NU* will be large; the code size for each micro-MMM might be big enough that full unrolling (even for small *NB*) might overflow the instruction cache.

6.4 Sensitivity to *Latency*

Latency is important only when the code generator is not using combined multiply-add instructions. SGI has a combined multiply-add, and the impact on performance of changes in the *Latency* parameter is less than 20 MFLOPS so we do not show it graphically.

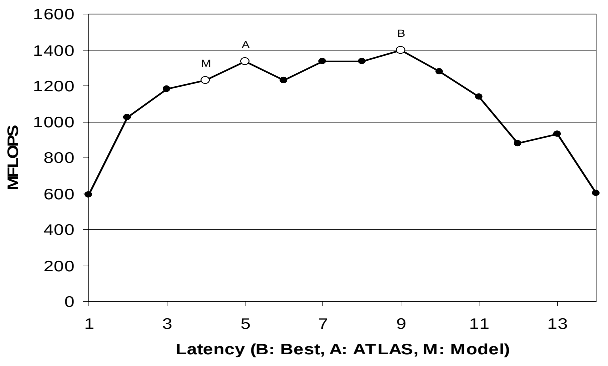


Figure 19: Performance vs. latency on Sun

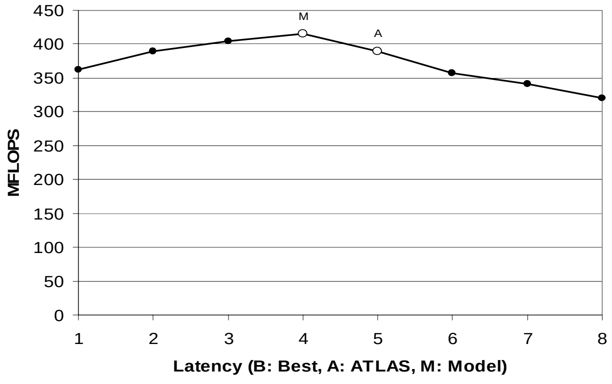


Figure 20: Performance vs. latency on Intel

We now consider machines without a multiply-add instruction. If *Latency* is too small, the multiplications do not finish in time for the corresponding additions, so the pipeline stalls. If the *Latency* is too big, the ATLAS code generator needs that many temporary registers, which limits the *MU* and *NU* unroll factors, reducing performance. Therefore we expect an inverse U shape for the graph of performance vs. *Latency*, which is what we see in Figure

19 and Figure 20. On the Sun, the Model chooses a sub-optimal value for *Latency*, which though close to the value ATLAS chooses, results in a 10% drop in performance.

6.5 Sensitivity to *Fetch* parameters

Our experiments showed that performance on the three machines is insensitive to the values of *FFetch*, *IFetch* and *NFetch* parameters.

6.6 Summary

Our experiments show that performance on all three machines is most sensitive to the register tiling parameters *MU* and *NU*. There are relatively few registers on most machines, so it is important to use them effectively. However, even for these sensitive parameters model-driven optimization was competitive. On the Sun, performance is also very sensitive to the *Latency* parameter. Paradoxically, although performance is relatively insensitive to cache tile size on machines with large L1 data caches, there appear to be some subtleties in choosing good values for this parameter, perhaps having to do with multiple memory hierarchy levels. Sub-optimal tile size and *Latency* choices contributed roughly equally to the 20% difference in performance between ATLAS-generated and Model-generated code on the Sun.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we compared the relative effectiveness of empirical and model-driven optimization in producing optimized BLAS libraries. To isolate the contribution of empirical optimization, we modified ATLAS so that it used optimization parameters derived from model-driven optimization, and compared the performance of code generated by the two approaches on three architectures.

A surprising conclusion from our experiments is that model-driven optimization can be very effective – on the SGI and Intel machines, performance of the generated code was very close to that of ATLAS-generated code. On the Sun, the model chose sub-optimal values for two parameters (tile size and *Latency*), and this led to a 20% difference in performance. It remains to be seen if more accurate modeling can eliminate this difference.

Empirical optimization is used in FFTW and SPIRAL to choose an optimal algorithm from a suite of algorithms, and not just to choose values for transformation parameters as in ATLAS. It is an open question whether model-driven optimization is effective in this context as well.

It would be interesting to see if empirical search can be speeded up by using modeling to bound the size of the search space.

Perhaps the most important conclusion of this work is that empirical search may not be necessary to generate high quality code, given the effectiveness of the model-driven approach. It is more difficult to determine what needs to be done to make compilers bridge the gap with library generators. Although all techniques used by ATLAS – loop tiling, unrolling, instruction scheduling etc. – have been part of the compiler writer's lore for many years, we cannot claim that it is easy to make compilers competitive with library generators. The developers of library generators know in advance the code to be generated and therefore can hardwire the search space and performance equations into their systems. With today's technology, it is feasible to automate the identification of the search space, but the development of performance equations – which were the main component of our strategy – is not well understood and cannot be automated at this time. There are, however, some promising results in this area [3].

A second perhaps equally important conclusion of this study is that there is still a significant gap in performance between the code generated by ATLAS and the BLAS routines. Although we do not understand the reason for this gap very well, it is clear that the problem of automating library generation remains open. The high cost of library and application tuning makes this one of the most important questions we face today.

APPENDIX A

In this appendix we present in pseudo-code an optimized mini-MMM code generated by ATLAS. The optimization parameter values chosen are *MU*=4, *NU*=2, *KU*=1; *IFetch*=*NFetch*=2; *FFetch*=1; *Latency*=2; and *MulAdd*=false.

To produce a compact representation, we define some notation.

- There will be *MU*=4 temporary registers devoted to A (rAi). LA_i will mean load i^{th} such register from L1D$.
- There will be *NU*=2 temporary registers devoted to B (rBj). LB_j will mean load j^t such register from L1D$.
- There will be *MU*NU* temporary registers devoted to C (rCij). S_{ij} will mean store register rCij into L1D$.
- $*_{ij}$ will mean: multiply rAi by rBj and store the value in a temporary register.
- $+_{ij}$ will mean: add the temporary value of the previous multiplication of rAi by rBj to rCij.

We start with mini-MMM code similar to Figure 4:

```
loop on N, step NU=2
  loop on M, step MU=4
    prefetch C;                    // FFetch=1
    loop K, step KU=1
      $LA_0$; $LB_0$;                        // IFetch=2
          $*_{00}$;
      $LA_1$; $LA_2$;                        // NFetch=2
                        $*_{10}$;
      $LA_3$; $LB_1$;                        // NFetch=2
          $+_{00}$;   $*_{20}$;   $+_{10}$;
          $*_{30}$;   $+_{20}$;   $*_{01}$;
          $+_{30}$;   $*_{11}$;   $+_{01}$;
          $*_{21}$;   $+_{11}$;   $*_{31}$;
          $+_{21}$;              $+_{31}$;
    end K
    $S_{00}$; $S_{10}$; $S_{20}$; $S_{30}$; $S_{01}$; $S_{11}$; $S_{21}$; $S_{31}$;
  end M
end N
```

Although this is not the final code ATLAS generates, from here we can make some important observations:

- As expected the main loop body contains *MU*NU*=4*2=8 multiplies and 8 corresponding adds. Each pair is *Latency*=2 FP instructions apart;
- Fetch parameters work on a per K-iteration basis, issuing IFetch loads from rAi and rBj, followed by several groups of NFetch loads with computation in-between.

The final transformation step ATLAS performs on the code is to software pipeline the K loop. The result looks as follows:

```
loop on N, step NU=2
  loop on M, step MU=4
    prefetch C;                 // FFetch=1
    // start the software pipeline
    $LA_0$;   $LB_0$;                 // IFetch=2
          $*_{00}$;
    $LA_1$;   $LA_2$;                 // NFetch=2
                          $*_{10}$;
    $LA_3$;   $LB_1$;                 // NFetch=2
    loop K, step KU=1
      // software pipeline pattern
          $+_{00}$;    $*_{20}$;    $+_{10}$;
          $*_{30}$;    $+_{20}$;    $*_{01}$;
          $+_{30}$;    $*_{11}$;    $+_{01}$;
          $*_{21}$;    $+_{11}$;    $*_{31}$;
      $LA_0$; $LB_0$;                 // IFetch=2
      $LA_1$; $LA_2$;                 // NFetch=2
          $+_{21}$;
          $*_{00}$;
      $LA_3$; $LB_1$;                 // NFetch=2
                          $+_{31}$;
                          $*_{10}$;
    end K
    // drain the software pipeline
          $+_{00}$;    $*_{20}$;    $+_{10}$;
          $*_{30}$;    $+_{20}$;    $*_{01}$;
          $+_{30}$;    $*_{11}$;    $+_{01}$;
          $*_{21}$;    $+_{11}$;    $*_{31}$;
          $+_{21}$;                 $+_{31}$;
    $S_{00}$; $S_{10}$; $S_{20}$; $S_{30}$; $S_{01}$; $S_{11}$; $S_{21}$; $S_{31}$;
  end M
end N
```

When *Latency* is large, ATLAS also tries to schedule the multiplies of the current iteration with the adds from the previous iteration and the data loads of the next iteration, effectively spanning three original iterations in the pipeline pattern.

8. REFERENCES

[1] R. Allan and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan Kaufman Publishing, 2002

[2] P. Boulet, A. Darte, T. Risset, and Y. Robert, *Pen-ultimate tiling?*, INTEGRATION, the VLSI Journal, Volume 17, pages 33-51, 1994

[3] G. Cascaval and D. Padua, *Estimating Cache Misses and Locality using Stack Distances*. To appear in the International Conference on Supercomputing (ICS), 2003

[4] P. Clauss, *Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs*, In proceedings of the International Conference on Supercomputing (ICS), 1996

[5] S. Coleman and K. S. McKinley. *Tile size selection using cache organization and data layout*. In proceedings of Programming Languages Design and Implementation (PLDI), 1995

[6] J. Dongarra and R. Schreiber, *Automatic blocking of nested loops*, Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee, May 1990

[7] B. B. Fraguela, R. Doallo, and E. Zapata: *Automatic Analytical Modeling for the Estimation of Cache Misses*. In proceedings of Parallel Architectures and Compilation Techniques (PACT), pages 221-231, 1999

[8] M. Frigo and S. G. Johnson. FFTW: *An adaptive software architecture for the FFT*. In proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), volume 3, pages 1381-1384, 1998

[9] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2 Edition, Morgan Kaufman, 1996

[10] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and H. A. G. Wijshoff . *Iterative compilation in program optimization*. In proceedings of Compilers for Parallel Computers, pages (CPC), pages 35-44, 2000

[11] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff. *A feasibility study in iterative compilation*. In proceedings of the International Symposium on High Performance Computing (ISHPC), 1999

[12] M. S. Lam, E. E. Rothberg, and M. E. Wolf. *The cache performance and optimizations of blocked algorithms*. In proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 63-74, 1999

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. *Evaluation techniques for storage hierarchies*. IBM Systems Journal 9, 2, 78-117, 1970

[14] A. C. McKellar and E. G. Coffman. *Organizing matrices and matrix operations for paged memory systems*. Communications of the ACM 12, 3, pages 153-165

[15] K. S. McKinley, S. Carr, and C. Tseng, *Improving data locality with loop transformations*, In ACM Transactions on Programming Languages and Systems, 18(4):424-453, 1996

[16] J. J. Navarro, T. Juan, and T. Lang, *MOB forms: a class of multilevel block algorithms for dense linear algebra operations*, In proceedings of the International Conference on Supercomputing (ICS), pages 354-362, 1994

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

[18] R. Whaley and J. Dongarra. *Automatically Tuned Linear Algebra Software*. Technical Report UT CS-97-366, LAPACK Working Note No.131, University of Tennessee, 1997

[19] M. Wolfe, *Iteration space tiling for memory hierarchies*, In SIAM Conference on Parallel Processing for Scientific Computing, 1987

[20] M. E. Wolf, D. E. Maydan, and D. Chen, *Combining loop transformations considering caches and scheduling*, In proceeding of the International Symposium on Microarchitecture (MICRO 29), pages 274-286, 1996

[21] J. Xiong, J. Johnson, R. Johnson, and D. Padua, *SPL: A Language and its Compiler for DSP Algorithms*, In proceedings of Programming Languages Design and Implementation (PLDI), 2001