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 empirical optimization is more effective than model-driven 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 dense numerical linear algebra libraries) with a model-based 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 although model-based optimization can be surprisingly effective, useful models may have to consider not only hardware parameters but also the ability of back-end compilers to exploit hardware resources.

1. INTRODUCTION

There is growing interest in the use of empirical program optimization for generating efficient code that is highly tuned to particular architectures. Unlike conventional compilers, which use architectural models to decide which transformations to apply and to choose parameters for particular transformations, a system that uses empirical optimization simply generates multiple program variants, runs them on the actual hardware, and chooses the one that performs best. Such a system does not need architectural models, so the code it generates is automatically tuned to the underlying hardware. Two well-known systems that employ empirical optimization are ATLAS [17] which generates highly-tuned linear algebra libraries and FFTW [7] which generates FFT libraries. Both these systems generate better code on a wide range of architectures than the native optimizing compilers.

What accounts for the relative success of empirical optimization when compared to conventional compiler technology? One possibility is that compilers are at a disadvantage because they are general-purpose and must optimize programs from any problem domain, whereas a system like ATLAS is a program-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 effectively performing certain optimizations that compilers do not know about. Yet another possibility is that these systems perform the same optimizations as compilers, but do them in a different order than compilers do (the so-called "phase-ordering problem"). Perhaps the architectural models used by compilers are overly simplistic, so compilers are unable to estimate key transformation parameters such as tile sizes accurately. To the best of our knowledge, no studies exist to provide clear answers to these questions.

This paper provides the first quantitative evaluation of the differences between empirical optimization and conventional model-driven optimization. To isolate the effects of empirical search, we built a modified version of ATLAS that determines transformation parameters not by empirical search but by using program and machine models. We then measured the performance of the code generated by both systems on a variety of architectures, and studied the generated code itself to understand performance differences.

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 using an almost exhaustive empirical search. In Section 4, we describe novel program and machine models which we use to estimate these parameters without doing any empirical searches. In Section 5, we compare both methods on a number of common high-performance platforms, measuring:

- the time spend to determine the parameter values,
- the values themselves, and
- the relative performance of resulting code.

We discuss future directions in Section 6.

2. HIGH-PERFORMANCE BLAS

In this section, we use the framework of restructuring compilers to describe how highly optimized code for the Basic Linear Algebra Subroutines (BLAS) is produced by ATLAS. The code produced depends on certain *optimization parameters*, which for now are assumed to be given to the code generator by an oracle.

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 α and β are 1.

It is straightforward to code this algorithm in a high-level language like C, as shown below.

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

Figure 1. Matrix Multiplication

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

It is well known that the code in Figure 1 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 1, 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 lowest level of the memory hierarchy. To use them effectively in matrix multiplication, it is necessary to register-allocate array values, which many compilers do not normally do. Finally, on in-order pipelined processors, 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. Although ATLAS is not a general-purpose restructuring compiler, the code it produces can be viewed as the result of performing the following sequence of code transformations on the high-level code on Figure 1.

- *Cache blocking*: The standard approach to exploiting data reuse in a perfectly-nested loop 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. ATLAS blocks only for the level 1 data cache (L1). Each of the small matrix multiplications multiplies an *MB*x*KB* matrix with a *KB*x*NB* matrix and accumulates the result in a *MB*x*NB* sub-matrix of *C*. In this paper, we call these operations mini-MMMs. The values of *MB, NB*, and *KB* are optimization parameters.

- *Register blocking*: The code for the mini-MMM in the previous step is itself blocked to make a better use of the general purpose registers. Each of the smaller matrix multiplications multiplies an *MU*x*KU* sub-matrix with a *KU*x*NU* sub-matrix and accumulates the result in a *MU*x*NU* sub-matrix of *C*. In this paper, we call these micro-MMM's. The micro-MMM's are implemented as a straight-line code segment that results from completely unrolling the three innermost loops. The values of *MU, NU*, and *KU* are optimization parameters.

- *Software pipelining*: Each mini-MMM essentially operates out of the L1 cache, so the most expensive operation in the innermost loop of a mini-MMM is the multiplication. To avoid stalling the processor, it is desirable to overlap a multiplication from one iteration with loads and additions from other iterations. This is accomplished by software pipelining the innermost loop of a mini-MMM. Obviously, ATLAS does not have a general-purpose software pipelining routine inside it. Instead, it uses three parameters called *IFetch*, *NFetch*, and *FFetch,* along with *MU*, *NU*, *KU*, and *Latency* (of multiplication) to software pipeline the computation. *IFetch* and *NFetch* determine how loads are interleaved with computation. *FFetch* is used to determine prefetech of C into registers. The roles of these parameters are explained in more detail in Figure 2. The values of *FFetch*, *IFetch*, *NFetch* and *Latency* are optimization parameters.

To put all this together, it is instructive to study the code generated by ATLAS for a mini-MMM, an example of which is shown in the Appendix A. This code corresponds to the following example schema:

```
compute loop stopping conditions
compute loop increments
  (for X=A, B and C & d=m, n, and k: incXd)
declare register double (rAi,rBj,rCij,ml)
do // N Loop
{
  do // M loop
  {
    prefetch C tile in L1 // if ffetch == 1
    start pipeline computation
      intermix with loads:
        first ifetch then groups of nfetch
    for (k = NB; k; k -= KU)
    {
      KU copies of
      | MU*NU copies of
      | | ml=rAi*rBj;
      | | rCij+=ml;
      | interleaved with blocs of nfetch loads
      | and one block of ifetch loads
      increment matrix pointers with incXk
    }
    drain pipeline computation
    increment matrix pointers with incXm
  }
  while (stop condition for M)
  increment matrix pointers with incXn
}
while (stop condition for N)
```

Figure 2: Schema for mini-MMM

One interesting and useful property of matrix multiplication is that the loops in the loop nest can be written in any order without affecting the outcome of the computation. Assuming a JIK loop order, the M loop corresponds to the I direction and the N loop corresponds to the J direction. ATLAS is implemented in C, so instead of performing array indexing, it uses pointer arithmetic to access arrays in order to deal with FORTRAN array layout.

First the loop termination conditions and loop variable increments are computed in terms of pointers into the arrays. Then several "register double" scalar variables are declared with the hope that the native compiler will allocate them to physical floating point registers. We declare *MU* such variables for *A*, naming them rAi (i=0,…*MU*-1); *NU* for *B*, naming them rBj (j=0,…*NU*-1) and *MU***NU* for *C*, naming them rCij (for the same values of i and j mentioned above).

If ATLAS determines that the machine has a multiply-add instruction that can be used profitably, then the body of the loop will contain the single statement `rCij+= rAi*rBj` instead of the pair `ml=rAi*rBj; rCij+=ml;`. Otherwise, ATLAS will make use of *TR* different ml (register) variables to improve the utilization of the multiply unit. Clearly, if there are *NR* registers available, all the ml, rAi, rBj, and rCij variables can be allocated in registers only when

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

The main computation happens in the body of the M loop. It has three distinct parts, which have to do with the software pipelining of the innermost (K) loop. The first part prefetches rCij into L1 (depending on the *FFetch* parameter) and schedules enough loads to rAi and rBj together with some computation to start the software pipeline. The second part is the K loop itself, which contains the repetitive pipelined pattern of intermixed loads and computation. The body of the loop is replicated *KU* times, effectively reducing the loop overhead. Finally, in the third part the pipeline is drained and *C* is updated with the values of rCij.

2.3 Versioning

As in many BLAS libraries, the library generated by ATLAS actually contains several versions of each high-level BLAS-3 algorithm such as the one shown in Figure 1. For example, it is often the case that *A*, *B*, and *C* are sub-matrices of larger matrices. In that case, it may be beneficial to copy these matrices into contiguous storage to minimize conflict misses during the execution of the operation. However, the overhead of copying may not be worthwhile if the matrix sizes are small, or we might be unable to copy for memory reasons. Most BLAS libraries therefore have both a copying and a non-copying version of each code; at runtime, the sizes of the matrices are used to determine which version should be executed. Decisions ATLAS makes at runtime include (i) whether to copy the tiles for mini-MMM into continuous memory, and (ii) loop order (JIK or IJK).

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 the following *MB*, *NB*, *KB*, *MU*, *NU*, *KU*, *Latency*, *IFetch*, *FFetch*, and *NFetch*.

3. HOW ATLAS FINDS PARAMETERS

ATLAS does its work in two phases. During the installation phase, ATLAS computes several *machine parameters*, such as cache size and number of registers. Using these values, it determines values for optimization parameters by empirical search. Values of machine parameters are used to restrict the search space as described below. Finally, it generates all the versions of mini-MMMs for the library. During the runtime phase, an application program calls ATLAS's general interface routine. The interface routine takes care of some trivial cases such as empty input matrices and the case when α=0 (which means the result is just βC). The routine then makes a sequence of calls to the appropriate mini-MMM to peform the matrix multiplication.

3.1 Estimating machine parameters

Since ATLAS is self-tuning, it does not require the user to provide the values of the 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, and
- the latency of the floating point unit.

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 [8].

Two other important architectural parameters are the L1 instruction cache size and the number of outstanding loads that the machine supports. These are not determined explicitly by ATLAS; 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 loop unrolling that is beneficial. Rather than estimate 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 their 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 *NBxNB* such that *NB* is a multiple of 4 that satisfies the formula:

$$16 \leq NB \leq 80; NB^2 \leq L1Size \qquad \textbf{(3)}$$

ATLAS only searches for square tiles, so *MB*=*KB*=*NB*. A "phase-ordering problem" in generating the mini-MMMs 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*). Further, ATLAS selects a suitable latency and fetch parameters for software pipelining of the K loop.

The *NB* that produces highest MFLOPS is chosen as "best *NB*" value and 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 Equation (2). The cases when *MU* or *NU* is 1 are treated specially. A test is performed to check whether 1x9 unrolling or 9x1 unrolling is better than 3x3 unrolling. If not, unrolling factors of the form 1xU and Ux1 are not checked for U>3.

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 problem 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 checks whether there exists a FP unit latency value that produces better performance than the hardware latency parameter. It checks all the values between 1 and 6, comparing MFLOPS and selecting the best one, using all the parameters as determined from the previous steps. It also ensures that the chosen value is multiple of *MU*NU*KU* to facilitate instruction scheduling in the software pipelining of the K loop.

3.2.5 Find Best Fetch

In this step ATLAS searches for the values of three different parameters explained in Section 2.2: *FFetch*, *IFetch* and *NFetch*. The respective search order and ranges are as follows: *FFetch*=0,1; *IFetch*=2..*MU*+*NU*; and *NFetch*=1..*MU*+*NU*-*IFetch*.

3.2.6 Find Non-Copy Version Crossover

The decision whether to copy or not to copy the original tiles into sequential memory locations is made at runtime. Therefore ATLAS needs to generate a non-copy mini-MMM version in case it is needed during execution. Because tiles can now be scattered throughout the memory, the probability of getting conflict misses in the L1 cache is increased. To reduce the probability of conflict misses, ATLAS uses a smaller block size. ATLAS starts searching from the copy version block size *NB* down, checking all candidates greater than 16 in steps of 4, until the performance deteriorates by 20%. The *NB* that yields highest performance in terms of MFLOPS is selected to be the optimal *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.

3.2.7 Find 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.

ATLAS takes two approaches to generating these cleanup codes. It can generate specialized version for some *L* between 1 and *NB*-1 or use a general version of the cleanup code.

The search process is as follows: starting with *L*=*NB*-1 and down with step 1, ATLAS generates a specialized version for this specific *L* and compares the performance of the specialized to the performance of the general version. 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.*

At runtime, ATLAS invokes the copy version whenever the collective size of the matrices is big enough to amortize the cost copying with computation, using the following formula:

$$M * N * K \leq NB^3 \qquad \textbf{(2)}$$

One other case in which ATLAS decides to use the generated non-copy version is for matrices larger than $(2^{22}-32)/(2^4*\mathrm{NB})$ in one dimension.

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 Finding *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 [5] 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. The solution is developed by modeling the problem as a constrained minimization problem. Darte et al. [2] suggests 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 [11]. This is generalized to determining rectangular tiles while minimizing capacity and conflict misses by Coleman and McKinley [4]. Ramanujam and Sadayappan have considered tiling in the context of distributed-memory computers. Wolf, Chen and Maydan [19] 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 [3]. Unfortunately, few if any of these papers have comparisons with hand-optimized BLAS code, so it is difficult to determine how useful these approaches are.

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 a mini-MMM in which the loops are executed in the JIK order (assuming the tiles have been copied to contiguous memory in advance):

```
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 for *NB*x*NB* Tiles and JIK Loop Order

We can distinguish between three different scenarios.

- Large Cache Model (LCM): This model is valid when the working-set of the mini-MMM fits into the level of cache we are optimizing for. In this model we only suffer cold misses, but we never suffer capacity misses, as nothing is ever evicted from cache.
- Small Cache Model (SCM): This model is valid when our cache is so small that it cannot even hold a single row of one of the tiles. In this case, any data reuse with more than NB intermediate element accesses will cause a capacity cache miss.
- Medium Cache Model (MCM): This model is valid when the cache is big enough to hold one (or several) rows of a tile, but is not enough to hold a full tile.

Our objective is to choose the largest *NB* that makes the LCM valid during the execution of the mini-MMM. 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, in each iteration of the outermost loop *j* we walk through the whole matrix *A*. In order to suffer only cold misses (and stay in LCM) it is evident that we need to be able to store A completely in the cache. This will require $MB*KB=NB^2$ cache lines.

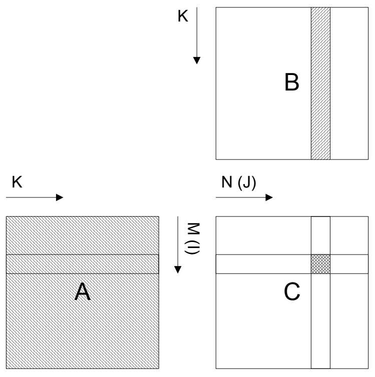

Figure 4. Matrix Indexing Scheme and Cache Usage for JIK

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 each 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.

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:

$$NB^2 + NB + 1 \le C \qquad \textbf{(3)}$$

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 (3), 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 storage.

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 (3) as follows:

$$\left\lceil \frac{NB^2}{B} \right\rceil + \left\lceil \frac{NB}{B} \right\rceil + 1 \le \frac{C}{B} \qquad \textbf{(4)}$$

The reasoning behind Inequality (4) 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. 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 (4) changes to the following:

$$\left\lceil \frac{NB^2}{B} \right\rceil + NB + 1 \le \frac{C}{B} \qquad \textbf{(5)}$$

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 will 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 framework for reasoning about such histories is the so-called "stack algorithm" [8]. 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. As a result, after each memory reference, the stack contains all the lines referenced so far by the program, ordered according to the most recent reference to each line. 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 our example from Figure 3 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:

$$A_{i,1}B_{1,j}A_{i,2}B_{2,j}\cdots A_{i,NB}B_{NB,j}C_{i,j} \qquad \textbf{(6)}$$

$C_{i,j}$ 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:

$$\begin{array}{l} A_{1,1}A_{1,2}\cdots A_{1,NB}C_{1,j} \\ \to A_{2,1}A_{2,2}\cdots A_{2,NB}C_{2,j} \\ \vdots \\ \to A_{NB-1,1}A_{NB-1,2}\cdots A_{NB-1,NB}C_{NB-1,j} \\ \to A_{NB,1}B_{1,j}A_{NB,2}B_{2,j}\cdots A_{NB,NB}B_{NB,j}C_{NB,j} \end{array} \qquad \textbf{(7)}$$

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 an optimal replacement cache, we needed only one element of *C*.

Continuing this argument, we see that we need space for two columns of *B* in the cache. This ensures that when we start working with the $j+2^{nd}$ column of *B*, the oldest items in the cache will be the elements of the j^{th} column.

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:

$$\left\lceil \frac{NB^2}{B} \right\rceil + 2\left\lceil \frac{NB}{B} \right\rceil + \left(\left\lceil \frac{NB}{B} \right\rceil + 1\right) \le \frac{C}{B} \quad \text{or}$$
$$\left\lceil \frac{NB^2}{B} \right\rceil + 3\left\lceil \frac{NB}{B} \right\rceil + 1 \le \frac{C}{B} \qquad \textbf{(8)}$$

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 the table above.

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 never fully associative. In spite of this, the models developed so far are relevant in many cases because tiles are usually copied into a sequential region of memory to avoid conflict misses during the execution of the mini-MMM.

In some cases ATLAS decides not to copy tiles. As mentioned before, this happens either when the cost of copying cannot be amortized with execution because the matrices are too small, or when the extra amount of memory needed for the copies is too large.

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 in [6] which uses a statistical approach to approximately predict the number of cache misses caused by a loop nest with constant bounds.

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.

If all three parameters (*MU*, *NU* and *KU*) are equal to some value *U*, we can use inequality (3) to constrain *U*:

$$U^2 + U + 1 \le NR \qquad \textbf{(9)}$$

Here NR is the number of floating point registers.

Because NR is usually small, it may be suboptimal 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, (9) is replaced with:

$$MU * NU + NU + 1 \leq NR \tag{10}$$

This means 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

Because of a peculiarity in the ATLAS code generator we actually need *MU* registers rather than just one register for *A*. In addition, 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 appropriatee constraint is given by Inequality (2):

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

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

$$NU^2 + 2NU + (TR - NR) \leq 0 \tag{11}$$

which we can solve for *NU*. Having obtained *NU*, we can solve (2) for *MU*:

$$MU = \frac{NR - TR - NU}{NU + 1} \tag{12}$$

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 Fetch Model

As we already saw in Sec. 2.2, ATLAS has three fetch parameters: *FFetch*, *IFetch* and *NFetch*. The most performance critical of these is *NFetch*, which corresponds to the number of outstanding loads (*OL*) the machine supports (which is a hardware parameter).

We choose *FFetch*=1 (which means to fully prefetch the portion of matrix C into registers) and *IFetch*=*NFetch*=*OL*

4.4 Summary

We have developed a model-based approach for choosing all optimization parameters used by ATLAS: *MB*, *NB*, *KB*, *MU*, *NU*, *KU*, *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 in which empirical optimization has been replaced with model-based optimization as described earlier. The comparison is performed on three different high-performance architectures with the following parameters:

Table 2. Test Platforms

	SGI	Sun	Intel
CPU	R12000	UltraSparcIII	PIII-Xeon
Frequency	270MHz	900MHz	550MHz
L1 Cache	32KB/32KB	64KB/32KB	16KB/16KB
L2 Cache	4MB		512KB
Memory	1GB	4GB	1GB
OS	IRIX64 v6.5	SunOS 5.8	RedHat 7.3
Compiler	MipsPRO F77 v7.3.1.1m	Workshop F77 v5.0	Intel ifc v6.0.1
Options	-O3 -64 -OPT:Olimit=15000 -TARG:platform=IP27	-dalign -native -xO5 -pad	-O3 -pad -unroll -tpp6 -xiKM

We compare both the installation time of the two versions, and the execution time of double-precision matrix multiplications of various sizes.

5.1 The Installation Phase

The installation phase can be divided into four parts. The first part, *Detecting Machine Parameters*, takes 6%-12% longer in the model-driven version of ATLAS, 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 the model-driven version performs this task in this part. The second part, *Estimating Optimization Parameters*, is where ATLAS performs the empirical search. In the model-driven version, this part takes almost no time because no search is performed; rather, the models are used to compute values for the optimization parameters. The final two parts of the installation phase generate the final source, and then compile it to make the library. There are no major differences between the two versions of ATLAS in the time they take to execute these two parts.

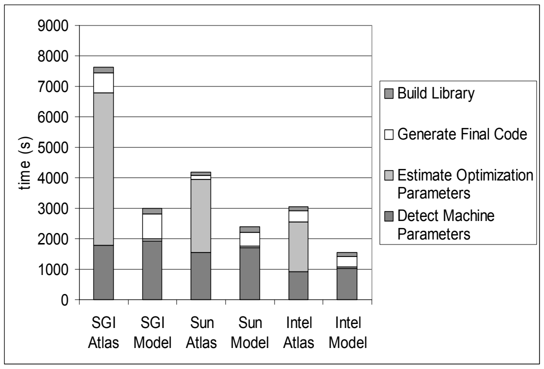

Figure 5. Installation Time

Table 3 and Table 4 show the values of the optimization parameters that are determined by ATLAS and by the model respectively. As is clear from this table, the values are quite close on the Intel and SGI machines, but on the SUN, the tile size and unroll factors are markedly different. We discuss these differences and their implications in more detail below.

Table 3. ATLAS Estimated Parameters

Architecture	Tile Size	Unroll Factors	Fetch Factors
	Copy / Non-Copy	MU / NU / KU	F / I / N
SGI	64 / 64	4 / 4 / 64	0 / 5 / 1
Sun	60 / 56	2 / 2 / 60	0 / 2 / 1
Intel	40 / 40	2 / 1 / 40	0 / 3 / 1

Table 4. Model Estimated Parameters

Architecture	Tile Size	Unroll Factors	Fetch Factors
	Copy / Non-Copy	MU / NU / KU	F / I / N
SGI	62 / 45	4 / 4 / 64	1 / 2 / 2
Sun	88 / 78	4 / 4 / 88	1 / 2 / 2
Intel	42 / 39	2 / 1 / 42	1 / 2 / 2

5.2 Comparison of Execution Time

In this section we compare the libraries generated by the original and the model-driven versions of ATLAS by comparing the execution times of both the mini-MMM routine (Table 5) and complete MMM for various matrix sizes (Figure 6, Figure 7 and Figure 8).

Table 5. Mini-MMM Performance Comparison

Architecture	ATLAS	Model	Difference
	(MFLOPS)	(MFLOPS)	(%)
SGI	457	453	0.9
Sun	738	365	51.5
Intel	394	384	2.5

From Table 5 we see that the performance of the two mini-MMMs is very similar on both the SGI and Intel machines but on the SUN, the code generated by the model-driven version is twice as slow as the one generated by ATLAS.

Next, we compare the performance of complete MMM using:

- mini-MMMs generated by ATLAS (with empirical search),
- mini-MMMs generated by model-driven optimization,
- 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 50 to 5000.

On the SGI machine, the best performer is the native BLAS library. The native compiler (MIPSPro) also does a good job of optimizing code. On the matrix sizes we tested our model is always within 2% of ATLAS in terms of MFLOPS performance. For the matrix sizes larger than 4000, the model outperforms ATLAS by roughly 30%, but both are much slower than BLAS, as ATLAS decides to use the Non-Copy version in this case.

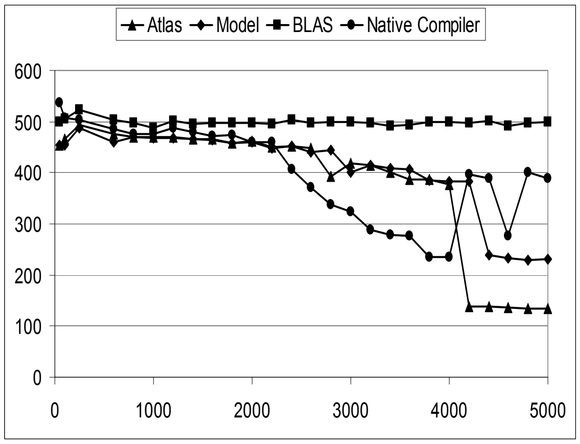

Figure 6. MMM Performance Comparison on SGI

The results for the Sun machine highlight some of the pitfalls of modelling. On this machine ATLAS showed remarkable performance, better even than the vendor provided BLAS library. Our model got somewhat close to the BLAS. If we compare the parameters used by the original version of ATLAS and by the model, we see two major reasons for the significant difference in performance.

- The model-driven version chose a much larger block size: it chose *NB*=88, compared to 60 for ATLAS. With *NB*=88, the working set fit into the 64KB L1 cache of the UltraSparc, but it occupies almost all of it and there is no room left for other data used by the code. When we changed *NB* to 84 there was a significant performance improvement.
- The model chose much bigger *MU/NU* unroll factors – 4x4, compared to the ATLAS detected values of 2x2. This problem arises from the fact that even though UltraSparc has 32 general-purpose registers, the native compiler uses only 15 of them. ATLAS determines this correctly, whereas the model-driven code tries to use all 32 registers, so the native compiler generates code with a lot of register spills. If we plug 15 in the model instead of 32, it correctly predicts the best unroll factors 2x2.

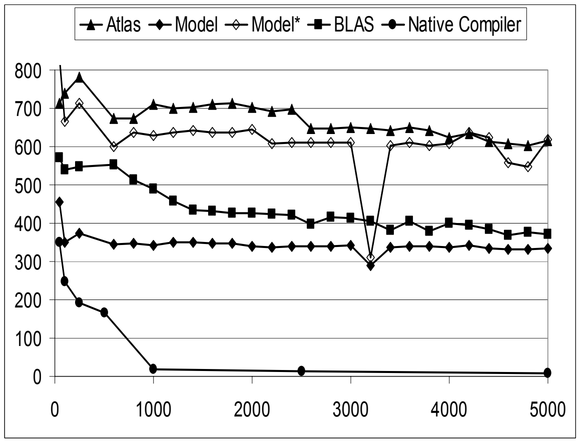

Figure 7. MMM Performance Comparison on Sun

The graph for UltraSparc shows one additional line (labeled Model*), which represents the performance of code generated with the improved model parameters. The performance difference with ATLAS is reduced to 7% on average.

One lesson from these results is that a model-driven optimization system cannot rely totally on its knowledge of machine parameters if it uses a back-end compiler like gcc to generate actual assembly code - the effectiveness with which the low-level compiler can exploit hardware resources can affect the choice of optimization parameters.

As on the SGI machine, the vendor-provided BLAS libraries perform best on the Intel machine. Both ATLAS and our model provide good performance, with the difference between them ranging from 3% to 10% (average is 7%). The performance obtained by using the native compiler is quite low because it does not do sany blocking.

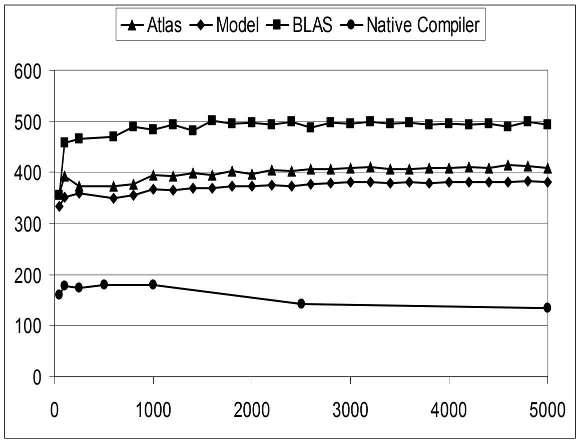

Figure 8. MMM Performance Comparison on Intel

6. CONCLUSIONS AND FUTURE WORK

In this paper, we compared the relative effectiveness of empirical optimization 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 very different machine architectures.

We found that model-driven optimization can be surprisingly effective – on the SGI and Intel machines, performance of the generated code was close to that of ATLAS-generated code. However, our model was not accurate for the SUN because it did not consider the limitations of the back-end compiler; therefore, the code produced was not competitive. One lesson from this experience is that a model-driven optimization system cannot rely totally on its knowledge of machine parameters if it uses a back-end compiler to generate actual assembly code.

In our current work, we are developing micro-benchmarks to estimate "effective" machine parameters that take limitations of the back-end compiler into account. We are also measuring the sensitivity of performance to the values of optimization parameters to understand where careful modeling is important. We are examining whether empirical search can be speeded up by using accurate models to bound the size of the search space. We will also investigate these issues for other problem domains such as the generation of FFT libraries.

APPENDIX A

In this appendix we will examine more closely an optimized mini-MMM code generated by ATLAS for MU=4, NU=2, KU=1; IFetch=NFetch=2; FFetch=1 and latency 2.

This example assumes that combined multiply-add instructions are not used. As we saw this is the case for all architectures we looked at.

In order to produce more 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.

Considering all these, we can give some preliminary view of the code generated in this case:

```
loop on N, step NU=2
  loop on M, step MU=4
    prefetch C;                         // FFetch=1
    loop K, step KU=1
      LA0;   LB0;                       // IFetch=2
             *00;
      LA1;   LA2;                       // NFetch=2
                             *10;
      LA3;   LB1;                       // NFetch=2
             +00;    *20;    +10;
             *30;    +20;    *01;
             +30;    *11;    +01;
             *21;    +11;    *31;
             +21;            +31;
    end K
    S00; S10; S20; S30; S01; S11; S21; S31;
  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, folloed 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
    LA0;     LB0;                       // IFetch=2
             *00;
    LA1;     LA2;                       // NFetch=2
                             *10;
    LA3;     LB1;                       // NFetch=2
    loop K, step KU=1
      // software pipeline pattern
             +00;    *20;    +10;
             *30;    +20;    *01;
             +30;    *11;    +01;
             *21;    +11;    *31;
      LA0;   LB0;                       // 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
```

One interesting point to take out from this is that *KU* is not an interesting parameter, because changing it just creates several copies of the body of the K-loop without trying to schedule instructions across the boundary between two copies.

Finally, here we observe that through pipelining we managed to schedule the computation of the current iteration together with the data loads of the next iteration. When we have sufficiently big latency, 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.

7. 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, *Penultimate tiling?*, INTEGRATION, the VLSI Journal, Volume 17, pages 33-51, 1994

[3] P. Clauss, *Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs*, In ICS'96, May, 1996

[4] S. Coleman and K.S. McKinley. *Tile size selection using cache organization and data layout*. In Proc. PLDI95. pages 279-290, 1995

[5] 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

[6] B. Fraguela, R. Doallo, E. Zapata: *Automatic Analytical Modeling for the Estimation of Cache Misses*. IEEE PACT 1999: 221-231

[7] M. Frigo and S. G. Johnson. FFTW: *An adaptive software architecture for the FFT*. Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, volume 3, pages 1381- 1384, Seattle, WA, May 1998

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

[9] T. Kisuki, P.M.W. Knijnenburg, M.F.P. O'Boyle, and H.A.G. Wijshoff . *Iterative compilation in program optimization*. In Proc. CPC2000, pages 35-44, 2000

[10] 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 Proc. ISHPC'99, 1999

[11] M.S. Lam, E.E. Rothberg, and M.E. Wolf. *The cache performance and optimizations of blocked algorithms*. In Proc. ASPLOS'91, PAGES 63-74, 1991

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

[13] A.C. McKellar and E.G. Coffman. *Organizing matrices and matrix operations for paged memory systems*. Commun. ACM 12, 3, 153-165

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

[15] Juan J. Navarro and Toni Juan and Tomas Lang, *MOB forms: a class of multilevel block algorithms for dense linear algebra operations*, ICS'94, pages 354-362, 1994

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

[17] 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

[18] M. Wolfe, *Iteration space tiling for memory hierarchies*, In 3rd SIAM Conference on Parallel Processing for Scientific Computing, December 1987

[19] M.E. Wolf, D.E. Maydan, and D. Chen, *Combining loop transformations considering caches and scheduling*, In MICRO 29, pages 274--286, Silicon Graphics, Mountain View, CA, 1996