Abstract

A major difference between compilers, and library generators such as ATLAS or FFTW is that compilers use simple architectural models to determine values for code optimization parameters such as tile sizes, whereas library generators use global search over the space of parameter values. Both ATLAS and FFTW produce better code than state-of-the-art compilers, and it is commonly believed that this is because the architectural abstractions used in compilers are too simplistic to permit accurate estimation of optimal values for code optimization parameters.

Recent work in the compiler community has shown that in fact, relatively simple models can be used to compute near-optimal values for these parameters. While this is adequate for most situations, a compiler that produces near-optimal code may not be good enough for generating libraries that must be highly tuned because they are used extensively by a large user community. In this paper, we argue that the right solution for this context is to use a combination of model refinement and local search. We demonstrate this by showing experimentally that a modified ATLAS system using model refinement and local search can produce BLAS code competitive with the code produced by the unmodified ATLAS system using global search.

1 Introduction

The compiler community has invested a lot of effort in inventing program optimization strategies which can produce efficient code from high-level programs, and which can scale to large programs and complex architectures. In spite of this, the quality of code produced by current compilers is abysmal compared to hand-written code, even for a simple kernel like matrix multiplication [15].

To address the inadequacies of current compilers, it is necessary to study applications for which highly tuned implementations are available; by comparing these implementations with compiler-generated code, we can understand what compilers are doing wrong. Benchmark suites like the SPEC and Perfect benchmarks are not suitable in this context, since there are no highly tuned implementations of these codes to serve as the "gold standard" for comparisons. Fortunately, such highly tuned implementations do exist in the domain of numerical linear algebra, which is why we have focused our efforts there.
The Basic Linear Algebra Subroutines (BLAS) are a key set of codes in dense numerical linear algebra. Most computer vendors provide carefully hand-tuned BLAS implementations, but the source code for their implementations is usually proprietary. Fortunately, Dongarra and his co-workers have implemented a portable BLAS generator called ATLAS [13], which is publicly available. Experiments show that the code produced by ATLAS performs nearly as well as native BLAS on many machines, so ATLAS is an excellent object of study for researchers interested in improving the sad state of the art of current compilers.

Why does ATLAS produce code that performs so much better than the code produced by current compilers? One major difference between current compilers and library generators like ATLAS [13] (and FFTW [4] and Spiral [6], which produce FFT code), is the approach they take to program optimization. Most compilers use simple architectural models of the hardware to determine whether an optimization is useful, and to estimate values for parameters associated with that optimization. When tiling a loop for example, a compiler may estimate the tile size using only the size of the data cache, ignoring its associativity and line size. The approach taken by library generators is very different, and we explain it with reference to the ATLAS system. This system has two key modules, shown in Figure 1.

1. **Code Generator**: For the purpose of this paper, this module generates an optimized matrix-multiplication routine, given certain **optimization parameters** as input. These optimization parameters are described in more detail in Section 2.1; intuitively, they tell the code generator what tile size to use ($N_B$), how much to unroll certain loops ($M_U, N_U, K_U$), etc.

2. **Empirical Search Engine**: To determine optimal values for these code optimization parameters, ATLAS employs a global search over the space of these parameters, performed under the direction of the Empirical Search Engine. This module enumerates each point in the parameter search space, and passes it on to the code generator, which produces the appropriate matrix-multiplication code (shown as mini-MMM code in Figure 1). This code is run on the actual machine, and its performance is recorded. Once the search is complete, the parameter values that give the best performance are used to generate the library. Section 2.2 describes this search in more detail.

To finish the global search in reasonable time, it is necessary to bound the search space. When ATLAS installs itself on a machine, it runs a set of micro-benchmarks to measure a set of **hardware parameters** such as the L1 data cache capacity [5], the number of registers, etc. These hardware parameters are used by the search engine to bound the search space for the optimization parameters. For example, the capacity of the L1 data cache is used to bound the search for $N_B$.

Experiments have shown that library generators produce much better code than general-purpose compilers do [4, 15]. It is commonly believed that this is because the architectural models used in compilers are too simplistic to permit accurate estimation of optimal values for optimization parameters. However, recent results of Yotov et al [15] have cast
doubt on this belief. To compare empirical optimization with model-based optimization, they built a modified version of ATLAS in which the empirical search engine was replaced with a module that used simple models to estimate values for optimization parameters. This model is described briefly in Section 3. Somewhat surprisingly, their experiments show that on the three architectures they considered, the performance of code produced by the model-driven version of ATLAS was within 10% to 20% of the performance of code produced by ATLAS using global search. Furthermore, the time required by the model-driven version was a small fraction of the time required to perform global search. These results suggest that the use of architectural abstractions in compilers need not come in the way of generating relatively high-quality code.

In this paper, we address two problems with model-driven optimization.

The first problem is to close the performance gap between code produced by library generators and code produced by using model-driven optimization. Although a compiler that quickly generates code that performs within 10% to 20% of highly tuned code is adequate in most situations, this performance penalty may be unacceptable for critical applications that will be run many times. Global search on the other hand does not scale well to large programs or to complex architectures.

The second problem is performance portability. It may seem that the use of global search ensures that library generators will work “out of the box” on new architectures, whereas model-driven optimization may fail if the model is a poor abstraction of the new architecture. However, global search is not a panacea. In particular, if the code generator does not exploit aspects of an architecture that are key to performance, the resulting code may be poor regardless of how exhaustive the search for optimal parameter values is. The methodology used in the ATLAS system to adapt to new architectures for which search alone is not sufficient is to include a collection of user-contributed hand-tuned kernels in the distribution; during the search process, the performance of these codes is evaluated, and if one of them performs better than the code generated by the code generator, it is used to produce the library. This methodology cannot be used for model-driven optimization because it runs counter to the spirit of using models to optimize programs.

The methodology that we advocate in this paper to both problems is to use a combination of model refinement and local search. To close the performance gap with code produced by empirical optimization, we advocate using local search in the neighborhood of the parameter values produced by using the model. Of course local search alone may not be adequate if the model is not a good abstraction of the architecture. In that case, we advocate using model refinement in the same spirit as ATLAS incorporates user-contributed code - we study the new architecture and refine the model as needed. Note that like the production of user-contributed code, model refinement must be done manually. Intuitively, in our methodology, small performance problems are tackled using local search, while large performance problems are tackled using model refinement.

Our experiments with ATLAS show that the combination of model refinement and local search is effective in closing performance gaps between the model-generated code and the code generated by global search, while at the same keeping library generation time small. However, it is important to realize that reducing library generation time is not the primary focus of our work; rather, our goal is to find optimization strategies for generating very high-performance code that can scale to large programs and complex architectures.

The rest of this paper is organized as follows. In Section 2, we describe how the ATLAS code generation module uses the code optimization parameters in code generation. We also describe the global search performed by ATLAS.
In Section 3, we briefly describe the model of Yotov et al [15] for computing values for transformation parameters. We have compared the performance of the code generated by the two approaches on about ten machines. In Section 3.2, we present some of these results that reveal potential for improvements both in the model and in ATLAS. In Section 4, we describe how model refinement and local search can be used to tackle these problems in the context of model-driven optimization. In Section 5, we present experimental results for the same machines as before, showing that this methodology addresses the performance problems identified earlier. In some cases, we obtain better code than is produced by the ATLAS code generator. We conclude in Section 6 with a discussion of future work.

2 Overview of ATLAS

Section 2.1 describes the code optimization parameters used by ATLAS. Section 2.2 describes the global search performed by ATLAS to find optimal values for these parameters.

2.1 Code optimization parameters

To explain the role of the code optimization parameters, we use the framework of restructuring compilers to describe the code generated by ATLAS (it is important to keep in mind that ATLAS is not a general-purpose restructuring compiler). We concentrate on matrix multiplication (MMM), which is the key routine in the BLAS. Naive MMM code is shown in Figure 2.

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

Figure 2: Naive MMM Code

2.1.1 Memory Hierarchy Optimizations

The code shown in Figure 2 can be optimized by tiling for the L1 data cache and registers.

- **Optimization for the L1 data cache:**
  To improve locality, ATLAS implements an MMM as a sequence of mini-MMMs, where each mini-MMM multiplies sub-matrices of size $N_B \times N_B$. $N_B$ is an optimization parameter whose value must be chosen so that the working set of the mini-MMM fits in the L1 cache.

  In the terminology of restructuring compilers, the triply-nested loop of Figure 2 is tiled with tiles of size $N_B \times N_B \times N_B$, producing an outer and an inner loop nest. For the outer loop nest, code for both the JIK and IJK loop orders are implemented. When the MMM library routine is called, it uses the shapes of the input arrays to decide which version to invoke. For the inner loop nest, only the JIK loop order is used, with $(j', i', k')$ as control variables. This inner loop nest multiplies sub-matrices of size $N_B \times N_B$, and we call this computation a mini-MMM.

- **Optimization for the register file:** ATLAS converts each mini-MMM into a sequence of micro-MMMs, where each micro-MMM multiplies an $M_U \times 1$ sub-matrix of A with a $1 \times N_U$ sub-matrix of B and accumulates the result into an $M_U \times N_U$ sub-matrix of C. $M_U$ and $N_U$ are optimization parameters that must be chosen so
that a micro-MMM can be executed out of the floating-point registers. For this to happen, it is necessary that \( M_U + N_U + M_U \times N_U \leq N_R \), where \( N_R \) is the number of floating-point registers.

In terms of restructuring compiler terminology, the \((j', i', k')\) loops of the mini-MMM from the previous step are tiled with tiles of size \( N_U \times M_U \times K_U \), producing an extra inner loop nest. The JIK loop order is chosen for the outer loop nest after tiling, and the KJI loop order for the inner loop nest.

The resulting code after the two tiling steps is shown in Figure 3. To keep this code simple, we have assumed that all step sizes in these loops divide the appropriate loop bounds exactly (so \( N_B \) divides \( M, N \), and \( K \), etc.). In reality, code should also be generated to handle the fractional tiles at the boundaries of the three arrays; we omit this clean-up code to avoid complicating the description. Figure 4 is a pictorial view of a mini-MMM computation within which a micro-MMM is shown using shaded rectangles.

```plaintext
// MMM loop nest (j, i, k)
// copy full A here
for j ∈ [1:N_B:M]
  // copy a panel of B here
  for i ∈ [1:N_B:N]
    // copy a tile of C here
    for k ∈ [1:N_B:K]
      // mini-MMM loop nest (j', i', k')
      for j' ∈ [j:N_U:j + N_B - 1]
        for i' ∈ [i:M_U:i + N_B - 1]
          for k' ∈ [k:K_U:k + N_B - 1]
            for k'' ∈ [k':1:k' + K_U - 1]
              // micro-MMM loop nest (j'', i'')
              for j'' ∈ [j':1:j' + N_U - 1]
                for i'' ∈ [i':1:i' + M_U - 1]
                  C_{i'', j''} = C_{i'', j''} + A_{i'', k''} * B_{k'', j''}

Figure 3: MMM tiled for L1 data cache and Registers
```

Figure 4: mini-MMM and micro-MMM

To perform register allocation for the array variables referenced in the micro-MMM code, the micro-MMM loop nest \((j'', i'')\) in Figure 3 is fully unrolled, producing \( M_U \times N_U \) multiply-add statements in the body of the middle loop nest. In the unrolled loop body, each array element is accessed several times. To enable register allocation of these array elements, ATLAS introduces a scalar temporary for each element of \( A \), \( B \), and \( C \) that is referenced in the unrolled micro-MMM code, and replaces array references in the unrolled micro-MMM code with references to these scalars. Appropriate assignment statements are introduced to initialize the scalars corresponding to \( A \) and \( B \) elements. In addition, assignment statements are introduced before and after the \( k' \) loop to initialize the scalars corresponding to \( C \) elements, and to write the values back into the array respectively. It is expected that the back-end compiler will
allocate floating-point registers for these scalars.

2.1.2 Pipeline scheduling

The resulting straight-line code in the body of the \( k' \) loop is scheduled to make better use of the processor pipeline. Note that the operations in the \( k' \) loop are the \( M_U + N_U \) loads of \( A \) and \( B \) elements required for the micro-MMM, and the corresponding \( M_U \times N_U \) multiplications and additions. On hardware architectures that have a fused multiply-add instruction, the scheduling problem is much simpler because multiplies and adds are executed together. Therefore, we only discuss the more interesting case when a multiply-add instruction is not present. An optimization parameter \( FMA \) tells the code generator whether to assume that a fused multiply-add exists. The scheduling of operations can be described as follows.

- Construct two sequences of length \( (M_U \times N_U) \), one containing the multiply operations and the other the add operations.
- Interleave the two sequences, to create a single sequence of the form \( \text{mul}_1 \text{ mul}_2 \ldots \text{mul}_{L_s} \text{ add}_1 \text{ mul}_{L_s+1} \text{ add}_2 \ldots \) that is obtained by skewing the adds by a factor of \( L_s \), where \( L_s \) is an optimization parameter. Intuitively, this interleaving separates most dependent multiplies and adds by \( 2 \times L_s - 1 \) other independent instructions to avoid stalling the processor pipeline.
- Inject the \( M_U + N_U \) loads of the elements of \( A \) and \( B \) into the resulting sequence of arithmetic operations by scheduling a block of \( I_F \) (Initial Fetch) loads in the beginning and blocks of \( N_F \) loads thereafter as needed. \( I_F \) and \( N_F \) are optimization parameters.
- Unroll the \( k' \) loop completely. The parameter \( K_U \) must be chosen to be large enough to reduce loop overhead, but not so large the body of the \( k' \) loop overflows the L1 instruction cache.
- Software pipeline the \( k' \) loop in such a way that operations from the current iteration are overlapped with operations from the previous iteration.

Note that skewing of dependent adds and multiplies increases register pressure; in particular, the following inequality must hold to avoid register spills:

\[
M_U \times N_U + M_U + N_U + L_s \leq N_R
\]  

2.1.3 Discussion

Table 1 lists the optimization parameters for future reference.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_B )</td>
<td>L1 data cache tile size</td>
</tr>
<tr>
<td>( M_U, N_U )</td>
<td>Register tile size</td>
</tr>
<tr>
<td>( K_U )</td>
<td>Unroll factor for ( k' ) loop</td>
</tr>
<tr>
<td>( L_s )</td>
<td>Latency for computation scheduling</td>
</tr>
<tr>
<td>( FMA )</td>
<td>1 if fused multiply-add, 0 otherwise</td>
</tr>
<tr>
<td>( F_F, I_F, N_F )</td>
<td>Scheduling of loads</td>
</tr>
</tbody>
</table>

Table 1: Summary of optimization parameters

There are a few details that we have omitted. In particular, ATLAS copies tiles of \( A, B, \) and \( C \) into sequential memory locations before performing the mini-MMM, if it thinks this would be profitable. The strategy for copying is
shown in Figure 3. ATLAS also incorporates a simple form of tiling for the L2 cache, called CacheEdge; we will not
discuss this because our focus is in the mini-MMM code, which is independent of CacheEdge.

It is intuitively obvious that the performance of the generated mini-MMM code suffers if the values of the optimiza-
tion parameters in Table 1 are too small or too large. For example, if $M_U$ and $N_U$ are too small, the $M_U \times N_U$ block
of computation instructions might not be large enough to hide the latency of the $M_U + N_U$ loads, and performance
suffers. On the other hand, if these parameters are too large, register spills will reduce performance. Similarly, if the
value of $K_U$ is too small, there is more loop overhead, but if this value is too big, the code in the body of the $k'$ loop
will overflow the instruction cache and performance will suffer. The goal therefore is to determine optimal values of
these parameters for obtaining the best mini-MMM code.

2.2 Global search in ATLAS

To find optimal values for the optimization parameters, ATLAS uses a global search strategy called orthogonal line
search [9]. Suppose we want to find the optimal value of a function $y = f(x_1, x_2, ... x_n)$. To find an approximate
solution, we reduce this $n$-dimensional optimization problem into a sequence of $1$-dimensional optimization problems
as follows: we order these parameters in some way and optimize them one at a time, using reference values for those
that have not yet been optimized. Orthogonal line search is an approximate method in the sense that it does not
necessarily find the optimal value of a function, but it might come close.

To specify an orthogonal line search, it is necessary to specify (i) the order in which the parameters are optimized,
(ii) the range of possible values considered during the optimization of each parameter, and (iii) the reference value
used for parameter $k$ during the optimization of parameters 1, 2, ..., $k-1$.

In ATLAS, the optimization sequence is the following: (i) $N_B$, (ii) $M_U$ and $N_U$, (iii) $K_U$, (iv) $L_s$ and (v) $F_F$, $I_F$, and $N_F$.

To find the best $N_B$, ATLAS generates a number of mini-MMMs for matrix sizes $N_B \times N_B$ where $N_B$ is a multiple
of 4 and $16 \leq N_B \leq \min(80, \sqrt{C_{L1}})$, where $C_{L1}$ is the capacity of the L1 data cache. During this search, the values
of $M_U$ and $N_U$ are set to the values closest to each other that satisfy Inequality (1) with $M_U > N_U$. For each block
size, ATLAS tries two extreme cases for $K_U$ no unrolling ($K_U = 1$) and full unrolling ($K_U = N_B$). Suitable $L_s$, $F_F$,
$I_F$, and $N_F$ are obtained from running the micro-benchmarks. The value of $N_B$ that produces highest MFLOPS is
chosen as “best $N_B$” 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.

To find the best $M_U$ and $N_U$, ATLAS tries all combinations of $M_U$ and $N_U$ that satisfy Inequality (1). For each
combination, the value of $N_B$ from the previous step and reference values for all other parameters are used to generate
mini-MMM code; the version that performs best determines the values chosen for $M_U$ and $N_U$.

The remaining parameters are optimized in a similar way. We omit the details because they are not relevant to
the rest of this paper.

3 Model-driven optimization

In this section, we summarize the model of Yotov et al [15] for estimating values for optimization parameters. This
model requires the following machine parameters.
• $C_{L1}$ and $B_{L1}$ – capacity and line size of the L1 data cache respectively
• $N_R$ – number of floating-point registers
• $L_h$ – latency in cycles of floating-point multiply
• $|ALU_{FP}|$ – number of floating-point pipes
• $FMA$ – existence of a fused multiply-add instruction

This is a superset of the machine parameters required by the ATLAS framework. Furthermore, modelling requires more accurate values for machine parameters than searching does; models use machine parameters to directly determine optimal values for optimization parameters, whereas search-based techniques use them only to bound the search space. Therefore, our system uses a more extensive and robust set of precise micro-benchmarks [1] than ATLAS does.

3.1 Estimating parameter values

3.1.1 Estimating $N_B$

There has been a lot of work in the compiler community on estimating tile sizes for general programs [2, 10, 14, 7]. In this section, we discuss a sequence of models of increasing accuracy for estimating tile sizes for matrix-multiplication.

If the three tiles of $A$, $B$, and $C$ are copied into sequential memory locations, we can avoid conflict and capacity misses if all three tiles fit in the L1 cache. This leads to the following inequality.

$$3 \times N_B^2 \leq C_{L1}$$ (2)

This is a simple model, but on most architectures, we can afford to tolerate some misses. If we assume that we can tolerate some conflict misses but still disallow capacity misses, the tile size can be made larger since it is not necessary to keep all three tiles in the cache for the duration of the mini-MMM computation. For example, for the $ijk$ loop order, we need to keep only one element of $C$ in cache, because $C$ is indexed by the control variables in the outermost two loops. For this loop order, we also need to cache the complete tile of $A$, since we walk the full tile for each iteration of the $j$ loop. Finally we need to cache a column of $B$, since the $j^{th}$ column is fully accessed for each iteration of the $i$ loop. Therefore, assuming that the cache has an optimal replacement policy, we obtain the following inequality:

$$N_B^2 + N_B + 1 \leq C_{L1}$$ (3)

Because caches have non-unit line size, it is not possible to keep just one element of $C$ in the cache; instead we must store an entire line. Correcting for the line size $B_{L1}$, we obtain the following inequality:

$$\left\lceil \frac{N_B^2}{B_{L1}} \right\rceil + \left\lceil \frac{N_B}{B_{L1}} \right\rceil + 1 \leq \frac{C_{L1}}{B_{L1}}$$ (4)

The replacement policy in real caches is usually pseudo-LRU. Intuitively, in case of LRU replacement, we need more cache so that unwanted blocks stay there until they become least recently used. This refinement leads to the following inequality [15]:

$$\left\lceil \frac{N_B^2}{B_{L1}} \right\rceil + 3 \left\lceil \frac{N_B}{B_{L1}} \right\rceil + 1 \leq \frac{C_{L1}}{B_{L1}}$$ (5)

Up to this point, we ignored the interaction between register tiling and L1 data cache tiling. Because of register
tiling, we actually deal with vertical panels of register tiles rather than columns of scalar elements. This consideration
leads to the final inequality used to compute \( N_B \) [15].

\[
\left\lceil \frac{N_B^2}{B_{L1}} \right\rceil + 3 \left\lceil \frac{N_B \times N_U}{B_{L1}} \right\rceil + \left\lceil \frac{M_U}{B_{L1}} \right\rceil \times N_U \leq \frac{C_{L1}}{B_{L1}} \tag{6}
\]

Once \( M_U \) and \( N_U \) are computed as described next, it is easy to compute \( N_B \) from this inequality.

### 3.1.2 Estimating \( M_U \) and \( N_U \)

Inequality (1), which is reproduced below for convenience, is used to determine \( M_U \) and \( N_U \). To use this inequality,
we need an estimate for \( L_s \), which is computed as described in Section 3.1.3.

\[
M_U \times N_U + M_U + N_U + L_s \leq N_R
\]

Initially, assume \( M_U = N_U \). This leads to Inequality (7), which can be solved for \( M_U \). This value can then be
substituted into Inequality 1 to determine \( N_U \).

\[
M_U^2 + 2 \times M_U + L_s - N_R \leq 0 \tag{7}
\]

Finally, these values are adjusted to ensure that \( M_U \) and \( N_U \) are at least 1 and \( M_U > N_U \).

### 3.1.3 Estimating \( L_s \)

From the discussion of instruction scheduling in Section 2.1, we see that a multiply operation in the innermost loop
body is separated from its dependent add by a total of \( 2 \times (L_s - 1) \) operations (\( L_s - 1 \) multiplies and \( L_s - 1 \) adds).

The value of \( L_s \) must be chosen so that add operations can be issued without waiting for the corresponding multiply
operations to complete. If the latency of multiplication is \( L_h \) cycles, and we have \( |ALU_{FP}| \) floating point units, it will
take \( \frac{2 \times (L_s - 1)}{|ALU_{FP}|} \) cycles to issue \( 2 \times (L_s - 1) \) floating point instructions. From this, Equation 8 follows.

\[
L_h = \frac{2 \times (L_s - 1)}{|ALU_{FP}|} \tag{8}
\]

We therefore obtain the following estimate for \( L_s \).

\[
L_s = \left\lceil \frac{L_h \times |ALU_{FP}|}{2} \right\rceil + 1 \tag{9}
\]

### 3.1.4 Estimating \( K_U \), FMA, \( F_F \), \( I_F \), and \( N_F \)

The parameter \( K_U \) is set so that the body of the kernel fits in the L1 instruction cache. In most cases, it is possible
to completely unroll the \( k' \) loop (\( K_U = N_B \)), without overflowing the L1 instruction cache.

The optimization parameter FMA is set to the corresponding value measured by the hardware micro-benchmarks.

Finally, performance is largely insensitive to the values of the fetch parameters, so they are set to \( (F_F, I_F, N_F) = (0, 2, 2) \).
3.2 Experimental results

We compared the performance of code generated by ATLAS and by the model-driven version of ATLAS on ten different architectures. For lack of space, in this section, we only discuss some of the more interesting performance results.

3.2.1 AMD Opteron 240

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core Frequency</td>
<td>1400 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 64 B/line</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64 KB, 64 B/line</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>1 MB, 64 B/line</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>8</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>4</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>No</td>
</tr>
<tr>
<td>Operating System</td>
<td>SuSE 9 Linux</td>
</tr>
<tr>
<td>C Compiler</td>
<td>GNU GCC 3.3</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>GNU Fortran 3.3</td>
</tr>
</tbody>
</table>

Table 2: AMD Opteron 240 Specifications

Table 2 shows the specifications of the AMD Opteron 240 machine we used in our experiments.

Figure 5: MMM Results for AMD Opteron 240

Figure 6: AMD Opteron 240 Sensitivity to $N_B$

On this platform, as well as to some extent on all other x86 CISC platforms, we observed a significant performance gap between the code generated using the model and the code generated by ATLAS (see the two lines labelled Model and Global Search in Figure 5). To understand the problem, we studied the optimization parameter values produced...
Figure 7: AMD Opteron 240 Sensitivity to $M_U$, $N_U$

by the two approaches. These values are shown in Table 5. The two sets of values are quite different, but that by itself is not necessarily significant. For example, Figure 6 shows how performance of the mini-MMM code changes as $N_B$ is changed, keeping all other parameters fixed. It can be seen that the values chosen by Global Search ($N_B = 60$) and Model ($N_B = 88$) are both good choices for that optimization parameter, even though they are quite different.

On the other hand, performance sensitivity to $M_U$ and $N_U$, shown in Figure 7, demonstrates that the optimal values of $(M_U, N_U)$ are $(6, 1)$. Notice that this graph is not symmetric with respect to $M_U$ and $N_U$, because an $M_U \times 1$ tile of $C$ is contiguous in memory, but a $1 \times N_U$ tile is not [12]. Global Search finds $(M_U, N_U) = (6, 1)$, whereas the model estimates $(2, 1)$. To clinch the matter, we verified that the performance difference disappears if $(M_U, N_U)$ are set to $(6, 1)$, and all other optimization parameters are set to the values estimated by the model.

Since the difference in performance between the code produced by global search and the code produced by using the model is about 40%, it is likely that there is a problem with the model presented in Section 3 for determining $(M_U, N_U)$. Evidence for this is provided by the line labelled Local Search in Figure 5, which shows there is significant performance gap even if we use a simple local search around the parameter values estimated by the model, which is described in more detail in Section 4.2.

In Section 4.1.1, we show how model refinement fixes this problem.

### 3.2.2 SUN UltraSPARC IIIi

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core Frequency</td>
<td>1060 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 32 B/line, 4-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>32 KB, 32 B/line, 4-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>1 MB, 32 B/line, 4-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>32</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>4</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>No</td>
</tr>
<tr>
<td>Operating System</td>
<td>SUN Solaris 9</td>
</tr>
<tr>
<td>C Compiler</td>
<td>SUN C 5.5</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>SUN FORTRAN 95 7.1</td>
</tr>
</tbody>
</table>

Table 3: SUN UltraSPARC IIIi Specifications

Table 3 shows the specifications of the SUN UltraSPARC IIIi machine we used in our experiments.

The optimization parameters derived by using the model and global search are shown in Table 7. Figure 14 presents the MMM performance results. On this machine, Model actually performs about 10% better than Global Search.
However, this platform is one of several that highlights a deficiency of the model that afflicts ATLAS Global Search as well. The problem lies in the choice of $N_B$. Figure 8 shows the sensitivity of performance to $N_B$. The values chosen by Global Search, Model and the best value (44, 84 and 208 respectively) are denoted by vertical lines.

![Figure 8: SUN UltraSPARC IIIi Sensitivity to $N_B$](image)

Initially, performance increases with increasing values for $N_B$. The first, slight performance drop, at $N_B = 208$ can be explained by applying the model in Inequality (2) but for the L2 cache ($C_{L2} = 1 MB$). This is the point at which all three tiles fit together in the L2 cache. The second, more pronounced, performance drop, at $N_B = 360$ can be explained by applying Inequality (6) for the L2 cache. After this point, L2 capacity misses start to occur and performance drops dramatically. Notice that there are no drops in performance around $N_B = 52$ and $N_B = 88$ which are the corresponding model-predicted values for the L1 data cache.

The problem is that both Model and Global Search perform tiling for the L1 cache, but it is more beneficial to tile for the L2 cache on this machine$^1$. In general, this is desirable if (1) the cache miss latency for the L1 data cache is close to that of the L2 cache, or (2) the cache miss latency of the L1 data cache is small enough that it is possible to entirely hide almost all of the L1 data cache misses with floating point computations.

We show how model refinement solves this problem in Section 4.1.2.

![Figure 9: SUN UltraSPARC IIIi Sensitivity to $M_U, N_U$](image)

$^1$In general, another possibility is to do multi-level cache tiling but the ATLAS code generator provides support for a single level only.
3.2.3 Intel Itanium 2

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core Frequency</td>
<td>1500 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>16 KB, 64 B/line, 4-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>16 KB, 64 B/line, 4-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>256 KB, 128 B/line, 8-way</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>3MB, 128B/line, 12-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>128</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>4</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>Yes</td>
</tr>
<tr>
<td>Operating System</td>
<td>RedHat Linux 9</td>
</tr>
<tr>
<td>C Compiler</td>
<td>GNU GCC 3.3</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>GNU Fortran 3.3</td>
</tr>
</tbody>
</table>

Table 4: Intel Itanium 2 Specifications

Table 4 shows the specifications of the Intel Itanium 2 machine we used in our experiments.

The values of optimization parameters estimated by using the model and global search are shown in Table 9. Figure 15 presents the MMM results.

![Figure 10: Intel Itanium 2 Sensitivity to NB](image)

Figure 10 shows the sensitivity of performance to $N_B$ on Intel Itanium 2. The values chosen by Model, Global Search, and the best value (30, 80 and 360 respectively) are denoted by vertical lines.

As on the SUN UltraSPARC IIIi, tiling for the L1 data cache is not beneficial. On this platform, even the L2 cache is “invisible” to $N_B$ sensitivity, and the drops in performance are explained by substituting the size of the L3 cache $C_{L3} = 3MB$ in Inequalities (2) and (6).
The second part of Figure 10 zooms into the interval $N_B \in [300, 400]$, which contains the value that achieves best performance ($N_B = 360$). As we can see, there are performance spikes and dips of as much as 300 MFlops. In particular, the value of $N_B = 362$ obtained by using Inequality (2) for the L2 cache is not nearly as good as that for $N_B = 360$. Values of $N_B$ that are divisible by $M_U$ and $N_U$ usually provide slightly better performance because there are no “edge effects”; that is, no special clean-up code needs to be executed for small left-over register tiles at the boundary. The number of conflict misses in the L1 and L2 caches can also vary with tile size.

Refining the model to account for effects like conflict misses is not likely to be tractable in general, so we advocate using local search around the model-predicted value, as discussed in detail in Section 4.2.1.

For completeness, we show the sensitivity of performance to $M_U$ and $N_U$ in Figure 11, although there is nothing interesting in this graph. Because of the extremely large number of registers on this platform ($N_R = 128$), the peak of the hill is more like a plateau with a multitude of good choices for the $M_U$ and $N_U$ unroll factors. Both Model and Global Search do well.

3.3 Summary

Our experiments point to two deficiencies with the model of Yotov et al [15]. On x86-based architectures like the Opteron, there is only a small number of registers, and the model does not choose $(M_U, N_U)$ optimally. On machines like the UltraSPARC IIIi and the Itanium 2, the model (and ATLAS Global Search) tile for the wrong cache level. Finally, local improvement seems useful for some parameters such as $N_B$ on the Itanium.

4 Closing the gap

We discuss model refinement in Section 4.1, and local search in Section 4.2.
4.1 Model refinement

4.1.1 Choosing $M_U$ and $N_U$ for small $N_R$

Recall that in Section 2, we discussed how operations within the innermost loop of the mini-MMM code are scheduled to obtain good performance even on machines without out-of-order execution. In particular, the separation between multiplies and their dependent additions increases register pressure; to avoid spills, Inequality (1), reproduced below for convenience, must hold.

$$M_U \times N_U + M_U + N_U + L_s \leq N_R \quad (10)$$

For the Opteron, Table 5 shows that the model chose $M_U = 2$, $N_U = 1$, $FMA = 0$, while Global Search chose $M_U = 6$, $N_U = 1$, $FMA = 1$. If instead the model had chosen, $M_U = 6$ and $FMA = 1$, while keeping the rest of the parameters the same, the mini-MMM performance rises to 2050 MFLOPS. The parameters values found by Global Search are puzzling for several reasons. First, the Opteron does not have a FMA instruction! Second, choosing 6 and 1 for the values of $M_U$ and $N_U$, violates Inequality (1) since the Opteron has only 8 registers. How can we explain this?

Recall that Inequality (1) should hold because ATLAS generates code which allocates an $M_U \times 1$ vector-tile of matrix $A$ (which we call $\bar{a}$), an $1 \times N_U$ vector-tile of matrix $B$ (which we call $\bar{b}$) and a $M_U \times N_U$ tile of matrix $C$ (which we call $\bar{c}$). It then performs the outer-product of $\bar{a}$ and $\bar{b}$ and accumulates the result into $\bar{c}$. The outer-product computation requires that each element of $\bar{a}$ be multiplied by each element of $\bar{b}$, and this reuse justifies storing these vectors in registers.

Notice that if $N_U = 1$, then $\bar{b}$ is a single scalar that is multiplied by each element of $\bar{a}$. Therefore no reuse exists for the elements of $\bar{a}$. This observation lets us generate the code in Figure 12, which uses 1 register for $\bar{b}$ ($rb$), 6 registers for $\bar{c}$ ($rc_1 \ldots rc_6$) and 1 temporary register ($rt$).

One might expect that this code will not perform well, as there are dependences between most of the adjacent instructions because of the temporary register $rt$. In fact the code in Figure 12 performs well because of two architectural features of the Opteron.

1. Out-of-order execution: it is possible to schedule several multiplications in successive CPU cycles without waiting for their corresponding adds to complete.

2. Register renaming: the single temporary register $rt$ is renamed to a different physical register for each pair of multiply-add instructions.

Performing instruction scheduling as described in Section 2 requires additional logical registers for temporaries, which in turn limits the sizes of the register tiles. If an architecture has a small number of logical registers but the processor implements out-of-order execution and register renaming, it is better to use the logical registers for allocating larger register tiles and leave instruction scheduling to the out-of-order hardware core which can use the extra physical registers to hold the temporaries.

These insights permit us to refine the model described in Section 3 as follows: for processors with a small number of logical registers and register renaming, set $N_U = 1$, $M_U = N_R - 2$, $FMA = 1$. Section 5 describes experimental results that demonstrate that this strategy eliminates the performance gap between the code produced by model-driven optimization and the code produced by ATLAS using global search.
Even if there are enough logical registers, this kind of scheduling may be beneficial if the ISA is 2-address rather than 3-address, because one of the operands is overwritten. This is true on the Opteron when the 16 SSE vector registers are used to hold scalar values, which is GCC’s default behavior. Even though Inequality 1 prescribes $3 \times 3$ register tiles, the refined model prescribes $14 \times 1$ tiles. Experiments show that this performs better [12].

Although there is a large body of existing work on register allocation and instruction scheduling for pipelined machines [16, 3, 8, 11], we are not aware of any prior work that has highlighted this peculiar interaction between compile-time scheduling and register allocation, and dynamic register-renaming and out-of-order execution.

### 4.1.2 Multilevel Memory Hierarchy

As discussed in Section 3.2, there are some machines for which tiling for the L2 or L3 cache will give better performance than tiling for the L1 cache. The model presented in Section 3 does not account for cache miss penalties at different cache levels, so although we estimate tile sizes for different cache levels, we cannot determine which level to tile for.

One approach to addressing this problem in the context of model-driven optimization is to refine the model to include miss penalties. Our experience however is that it is difficult to use micro-benchmarks to measure miss penalties accurately for lower levels of the memory hierarchy on modern machines. Therefore, we decided to estimate tile sizes for all the cache levels according to Inequalities (2) and (6), and then empirically determine which one gives the best performance.

Notice that in the context of global search, the problem can be addressed by making the search space for $N_B$ large enough. However, this would increase the search time substantially since the size of an L3 cache, which would be used to bound the search space, is typically much larger than the size of an L1 cache. This difficulty highlights the advantage of our approach of using model-driven optimization together with a small amount of search - we can tackle multi-level memory hierarchies without increasing installation time significantly.
4.2 Local search

In this section, we describe how local search can be used to improve the $N_B$, $M_U$, $N_U$, and $L_s$ optimization parameters chosen by the model.

4.2.1 Local Search for $N_B$

If $N_{BM}$ is the value of $N_B$ estimated by the model, we can refine this value by local search in the interval $[N_{BM} - \text{lcm}(M_U, N_U), N_{BM} + \text{lcm}(M_U, N_U)]$. This ensures that we examine the first values of $N_B$ in the neighborhood of $N_{BM}$ that are divisible by both $M_U$ and $N_U$.

4.2.2 Local Search for $M_U$, $N_U$, and $L_s$

Unlike sensitivity graphs for $N_B$, sensitivity graphs for $M_U$ and $N_U$ tend to be convex in the neighborhood of model-predicted values. This is probably because register allocation is under compiler control, and there are no “conflict misses”. Therefore, we use a simple hill-climbing search strategy to improve these parameters.

We start with the model predicted values for $M_U$, $N_U$, and $L_s$ and determine if performance improves by changing each of them by +1 and −1. We continue following the path of increasing performance until we stop at a local maximum. On platforms on which there is a Fused-Multiply-Add instruction ($FMA = 1$), the optimization parameter $L_s$ has no effect on the generated code and in that case we only consider $M_U$ and $N_U$ for the hill-climbing local search.

5 Experimental Results

We evaluated the following six approaches on a large number of modern platforms, including DEC Alpha 21264, IBM Power 3/4, SGI R12000, SUN UltraSPARC IIIi, Intel Pentium III/4, Intel Itanium 2, AMD Athlon MP, and AMD Opteron 240.

1. Model: This approach uses the model of Yotov et al [15] as described in Section 3.
2. Refined Model: This approach uses the refined model for $(M_U, N_U)$ described in Section 4.1.1.
3. Local search: This approach uses local search as described in Section 4.2, in the neighborhood of parameter values determined by Refined Model.
4. Multi-level Local Search: This approach is the same as Local Search, but it considers tiling for lower levels of the memory hierarchy as described in Section 4.1.2.
5. Global Search: This is the ATLAS search strategy.
6. Unleashed: This is the full ATLAS distribution that includes user-contributed code, installed with accepting all defaults that the ATLAS team have provided. As such, it usually performs optimizations which are not exposed through the ATLAS’ Code Generator and therefore it is normally not directly comparable to our results.

For lack of space, we present results only for the machines discussed earlier in this paper.

5.1 AMD Opteron 240

Table 5 shows the values of the optimization parameters for Model, Refined Model, Local Search, Multi-Level (ML) Local Search, Global Search, and Unleashed, along with the corresponding performance numbers for mini-MMM.
Table 5: Optimization Parameters for AMD Opteron 240

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Machine</th>
<th>Optimization</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>101</td>
<td>2</td>
<td>103</td>
</tr>
<tr>
<td>Refined Model</td>
<td>101</td>
<td>2</td>
<td>103</td>
</tr>
<tr>
<td>Local Search</td>
<td>101</td>
<td>31</td>
<td>132</td>
</tr>
<tr>
<td>ML Local Search</td>
<td>101</td>
<td>126</td>
<td>227</td>
</tr>
<tr>
<td>Global Search</td>
<td>148</td>
<td>375</td>
<td>523</td>
</tr>
<tr>
<td>Unleashed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Timings for AMD Opteron 240

Figure 13 shows the MMM performance for all approaches. Local Search and Multi-Level Local Search are not plotted on this platform, because as Table 5 suggests, their performance is virtually equivalent to that of Refined Model.

The MMM performance achieved by Model + Local Search is only marginally worse than that of Global Search, which according to our sensitivity analysis is due to a slightly suboptimal value of $N_B$. Had we extended the interval in which we do local $N_B$ search by a small amount, we would have achieved the same performance.

In summary, the model refinement described in Section 4.1.1 to take into account the small number of logical registers on this machine is sufficient to address performance problems with basic model of Yotov et al [15].

Figure 14 shows the MMM performance for all approaches. Refined Model and Local Search are not plotted on this platform, because as Table 7 suggests, their performance is virtually equivalent to that of Model.

5.2 SUN UltraSPARC IIIi

Table 7 shows the values of the optimization parameters for Model, Refined Model, Local Search, Multi-Level (ML) Local Search, Global Search, and Unleashed, along with the corresponding performance numbers for mini-MMM.

Table 8 shows the times taken by our micro-benchmarks and by the ATLAS micro-benchmarks for determining machine and optimization parameters.

Figure 14 shows the MMM performance for all approaches. Refined Model and Local Search are not plotted on this platform, because as Table 7 suggests, their performance is virtually equivalent to that of Model.
Table 7: Optimization Parameters for SUN UltraSPARC IIIi

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Machine</th>
<th>Optimization</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>112</td>
<td>7</td>
<td>119</td>
</tr>
<tr>
<td>Refined Model</td>
<td>112</td>
<td>7</td>
<td>119</td>
</tr>
<tr>
<td>Local Search</td>
<td>112</td>
<td>127</td>
<td>239</td>
</tr>
<tr>
<td>ML Local Search</td>
<td>112</td>
<td>496</td>
<td>608</td>
</tr>
<tr>
<td>Global Search</td>
<td>203</td>
<td>1233</td>
<td>1436</td>
</tr>
</tbody>
</table>

Table 8: Timings for SUN UltraSPARC IIIi

Model performs marginally better than Global Search because the ATLAS micro-benchmarks estimated that the L1 data cache size is 16KB, rather than 64 KB. This overly restricted the \( N_B \) interval examined by Global Search, leading to poor performance.

Multi-Level Local Search performs better than Local Search because it finds that it is better to tile for the L2 cache rather than for the L1.

Figure 14: MMM Results for SUN UltraSPARC IIIi

5.3 Intel Itanium 2

The description of this platform was presented in Table 4.

Table 9 shows the values of the optimization parameters for Model, Refined Model, Local Search, Multi-Level (ML) Local Search, Global Search, and Unleashed, along with the corresponding performance numbers for mini-MMM.

Table 9: Optimization Parameters for Intel Itanium 2

Table 10 shows the times taken by our micro-benchmarks and by the ATLAS micro-benchmarks for determining machine and optimization parameters.
### Table 10: Timings for Intel Itanium 2

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Machine</td>
<td>Optimization</td>
</tr>
<tr>
<td>Model</td>
<td>143</td>
</tr>
<tr>
<td>Refined Model</td>
<td>143</td>
</tr>
<tr>
<td>Local Search</td>
<td>143</td>
</tr>
<tr>
<td>ML Local Search</td>
<td>143</td>
</tr>
<tr>
<td>Global Search</td>
<td>1554</td>
</tr>
</tbody>
</table>

Figure 15: MMM Results for Intel Itanium 2

Figure 15 shows the MMM performance for all these approaches. Refined Model and Local Search are not plotted on this platform, because as Table 9 suggests, their performance is virtually equivalent to that of Model.

Model does not perform well because it tiles for the L1 data cache. For the Itanium, ATLAS used the size of the L2 cache (256KB) to restrict $N_B$, effectively selecting the maximum value in the search range ($N_B = 80$). Nevertheless this tile size is not optimal either. The Multi-level model determined that tiling for the 3 MB L3 cache is optimal, and chooses a value of $N_B = 362$. This is refined to $N_B = 360$ by local search. This improves performance compared to both Model and Global Search.

### 5.4 SGI R12000

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Core Frequency</td>
<td>270 MHz</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32 KB, 32 B/line, 2-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>32 KB, 32 B/line, 2-way</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>4 MB, 32 B/line, 1-way</td>
</tr>
<tr>
<td>Floating-Point Registers</td>
<td>32</td>
</tr>
<tr>
<td>Floating-Point Functional Units</td>
<td>2</td>
</tr>
<tr>
<td>Floating-Point Multiply Latency</td>
<td>2</td>
</tr>
<tr>
<td>Has Fused Multiply Add</td>
<td>No</td>
</tr>
<tr>
<td>Operating System</td>
<td>IRIX64</td>
</tr>
<tr>
<td>C Compiler</td>
<td>SGI MIPSPro C 7.3.1.1m</td>
</tr>
<tr>
<td>Fortran Compiler</td>
<td>SGI MIPSPro FORTRAN 7.3.1.1m</td>
</tr>
</tbody>
</table>

Table 11: SGI R12000 Specifications

Table 11 shows the specifications of the SGI R12000 machine we used in our experiments.

Table 12 shows the values of the optimization parameters for Model, Refined Model, Local Search, Multi-Level (ML) Local Search, Global Search, and Unleashed, along with the corresponding performance numbers for mini-MMM.

Table 13 shows the times taken by our micro-benchmarks and by the ATLAS micro-benchmarks for determining machine and optimization parameters.

Figure 16 shows the MMM performance on the SGI R12K. Refined Model and Local Search are not plotted on this platform, because as Table 12 suggests, their performance is virtually equivalent to that of Model.
Table 12: Optimization Parameters for SGI R12000

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Machine</th>
<th>Optimization</th>
<th>Total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>58</td>
<td>118</td>
<td>13</td>
</tr>
<tr>
<td>Refined Model</td>
<td>58</td>
<td>118</td>
<td>131</td>
</tr>
<tr>
<td>Local Search</td>
<td>58</td>
<td>118</td>
<td>457</td>
</tr>
<tr>
<td>ML Local Search</td>
<td>418</td>
<td>118</td>
<td>496</td>
</tr>
<tr>
<td>Global Search</td>
<td>64</td>
<td>251</td>
<td>2131</td>
</tr>
</tbody>
</table>

Table 13: Timings for SGI R12000

Figure 16: MMM Results for SGI R12000

The most interesting fact on this platform is that Multi-Level Local Search successfully finds that it is worth tiling for the L2 cache. By doing this, it achieves better performance than even the native BLAS.

Global Search achieves slightly better performance than Model due to the minor differences in several optimization parameters. Unleashed does fine for relatively small matrices but for large ones performs worse than Model and Global Search. Although not entirely visible from the plot, on this platform, the native compiler (SGI MIPSPro) does a relatively good job.

6 Conclusions and Future Work

The compiler community has invested considerable effort in inventing program optimization strategies which can produce high-quality code from high-level programs, and which can scale to large programs and complex architectures. In spite of this, current compilers produce very poor code even for a simple kernel like matrix multiplication. To make progress in this area, we believe it is necessary to perform detailed case studies.

This paper reports the results of one such case study. Previously, Yotov et al [15] have shown that model-driven optimization can produce BLAS codes with performance within 10-20% of that of code produced by empirical optimization. We have shown that this remaining performance gap can be eliminated by a combination of model refinement and local search. The model refinement (i) corrects the instruction scheduling strategy for machines on which there are relatively few logical registers, and (ii) opens up the possibility of tiling for lower levels of the memory hierarchy. On some machines, this gave better performance than both ATLAS Global Search and the native BLAS.
We believe that this combination of model refinement and local search is promising, and it is the cornerstone of a system we are building for generating BLAS libraries that are optimized for many levels of the memory hierarchy, a problem for which global search is not tractable.

References


