Think Globally, Search Locally*

Kamen Yotov, Keshav Pingali, Paul Stodghill,
{kyotov,pingali,stodghil}@cs.cornell.edu
Department of Computer Science,
Cornell University,
Ithaca, NY 14853.

ABSTRACT

A key step in program optimization is the determination of optimal values for code optimization parameters such as cache tile sizes and loop unrolling factors. One approach, which is implemented in most compilers, is to use analytical models to determine these values. The other approach, used in library generators like ATLAS, is to perform a global empirical search over the space of parameter values.

Neither approach is completely suitable for use in general-purpose compilers that must generate high quality code for large programs running on complex architectures. Model-driven optimization may incur a performance penalty of 10-20% even for a relatively simple code like matrix multiplication. On the other hand, global search is not tractable for optimizing large programs for complex architectures because the optimization space is too large.

In this paper, we advocate a methodology for generating high-performance code without increasing search time dramatically. Our methodology has three components: (i) modeling, (ii) local search, and (iii) model refinement. We demonstrate this methodology by using it to eliminate the performance gap between code produced by a model-driven version of ATLAS described by us in prior work, and code produced by the original ATLAS system using global search.

1. INTRODUCTION

A key step in performance optimization is the determination of optimal values for code optimization parameters like cache tile sizes and loop unrolling factors. One approach, which is implemented in library generators like ATLAS [1], FFTW [8] and SPIRAL [12], is to perform an empirical search over a space of possible parameter values, and select the values that give the best performance. An alternative approach, implemented in most compilers, is to use analytical models, parameterized by the values of key hardware parameters such as the capacity of the cache and the number of registers, to estimate optimal parameter values [2, 4, 14].

In practice, neither approach is completely satisfactory, particularly in the context of compilers. Empirical global search can take a very long time, especially for complex programs and complex architectures, since the size of the optimization space can be very large. To address this problem, some researchers are investigating more advanced search algorithms such as the simplex method [6], but it remains to be seen if this approach is effective in reducing search time substantially, without reducing the quality of the produced code. Model-driven optimization on the other hand may result in performance penalties of 10-20% even for a relatively simple code like matrix multiplication [17], which may be unacceptable in some contexts.

In this paper, we explore a different approach that we believe combines the advantages of both approaches. It is based on (i) modelling, (ii) local search, and (iii) model refinement. We advocate the use of models to quickly estimate optimal parameter values (it is important to realize that reducing library generation time is not the focus of our work; rather, it is to find optimization strategies for generating very high-performance code that can be used in general-purpose compilers, because they scale to large programs and complex architectures). However, all useful models are abstractions of the underlying machine; for example, it is difficult to model conflict misses, so most cache models assume a fully-associative cache. Therefore, it is possible that a particular model may omit some features relevant to performance optimization for some code. To close the performance gap with code produced by exhaustive 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 a poor abstraction of the underlying machine. In that case, we advocate using model refinement - we study the architecture and refine the model appropriately. Intuitively, in our approach, small performance gaps are tackled using local search, while large performance gaps are tackled using model refinement.

The experiments reported in this paper use a modified version of the ATLAS system that implements this methodology. They show that the combination of model refinement and local search can be effective in closing performance gaps between the model-generated code and the code generated by global search, while keeping code generation time small.

The rest of this paper is organized as follows. In Section 2, we describe the optimization parameters used in the ATLAS system, and the global search process used by AT-

*This work was supported by an IBM Faculty Partnership Award, DARPA Grant NBCH30390004, NSF grants RI-9972853, ACI-0085969, ACI-012140, ACI-0406345, and ACI-0509324.

ICS'05, June 20–22, 2005, Boston, MA, USA

LAS to find optimal values for these parameters. We also summarize an analytical model [17] for estimating optimal values of these optimization parameters. In Section 3, we discuss experimental results on a number of machines that reveal the potential for improvements in this model (and in ATLAS). In Section 4, we describe how model refinement and local search can be used to tackle these performance problems. In Section 5, we present experimental results for the same machines as before, showing that this methodology addresses the performance problems identified earlier. In fact, in most cases, we actually obtain better code than is produced by the ATLAS code generator. We conclude in Section 6 with a discussion of future work.

2. ATLAS

Figure 1 shows a block diagram of the ATLAS system[1]. There are two main modules: *Code Generator*, and *Search Engine*.

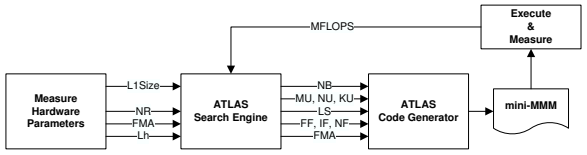


Figure 1: Empirical Optimization Architecture

For the purpose of this paper, the *Code Generator*, given certain *optimization parameters* which are described in more detail in Section 2.1, produces an optimized matrix-multiplication routine, shown as mini-MMM in Figure 1. This routine is C code which must be compiled using the native C compiler to produce an executable.

The *Search Engine* determines optimal values for these optimization parameters by performing a global search over the space of values for these parameters. Values corresponding to each point of the search space are passed to the code generator and the resulting mini-MMM 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. The search process is described in more detail in Section 2.2.

To finish the search in reasonable time, it is necessary to bound the search space. When ATLAS is installed on a machine, it executes a set of micro-benchmarks to measure certain *hardware parameters* such as the L1 data cache capacity [13], the number of registers, etc. The search engine uses these hardware parameters to bound the search space for the optimization parameters.

2.1 Optimization parameters

Although ATLAS is not a general-purpose compiler, it is useful to view the output of its code generator as if it were the result of applying restructuring compiler transformations to the naïve matrix multiplication code shown in Figure 2. A detailed description is given in [17]. Here we provide a summary of the relevant points.

To improve locality, ATLAS decomposes an MMM into a sequence of *mini-MMM*s, where each mini-MMM multiplies

[1]The complete ATLAS system includes hand-written code for various routines, which may be used to produce the library on some machines if it is found to outperform the code produced by the ATLAS code generator. Here, we only consider the code produced by the ATLAS code generator.

```
for i ∈ [0 : 1 : N − 1]
    for j ∈ [0 : 1 : N − 1]
        for k ∈ [0 : 1 : N − 1]
            C_{i,j} ← C_{i,j} + A_{i,k} × B_{k,j};
```

Figure 2: Naïve MMM Code

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 $\langle i, j, k\rangle$ loop nest in Figure 2 is tiled with tiles of size $N_B \times N_B \times N_B$, resulting in two new loop nests – *outer* and *inner*. ATLAS uses either $\langle j, i, k\rangle$ or $\langle i, j, k\rangle$ for the loop order of the outer loop nest, depending on the shape of the matrix arguments. The inner loop nest, which constitutes the mini-MMM, is always in the $\langle j', i', k'\rangle$ loop order.

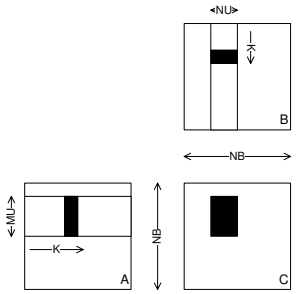


Figure 3: mini-MMM and micro-MMM

Each mini-MMM is further decomposed into a sequence of *micro-MMM*s, as shown in Figure 3, 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.

In the terminology of restructuring compilers, the mini-MMM loop nest is tiled with tiles of size $N_U \times M_U \times K_U$, resulting in a new $\langle k'', j'', i''\rangle$ loop nest (which is always executed using this loop order).

The resulting code after these two tiling steps is shown in Figure 4. To keep this code simple, we have assumed that all loop step sizes evenly divide the corresponding loop bounds exactly. In reality, code should also be generated to handle the fractional tiles at the boundaries of arrays; we omit this *clean-up* code to avoid complicating the description.

```
// full MMM ⟨j, i, k⟩
// copy the full matrix A
for j ∈ [1 : N_B : M]
    // copy a panel of B
    for i ∈ [1 : N_B : N]
        // possibly copy a tile of C
        for k ∈ [1 : N_B : K]
            // mini − MMM ⟨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 ⟨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 4: MMM tiled for cache and registers

To perform register allocation, the micro-MMM loop nest $\langle j'', k''\rangle$ is fully unrolled and the referenced array variables

are scalarized. Once the code for loading elements of C is lifted outside the k'' loop, the body of this loop contains $M_U + N_U$ loads and $M_U \times N_U$ multiply-add pairs. ATLAS schedules this basic block based on the FMA, L_s, I_F, and N_F optimization parameters as follows.

1. Intuitively, FMA is 0 if code should be generated without assuming that the hardware supports a fused multiply-add. In that case, dependent multiply and add operations are separated by L_s other multiplies adds. This produces sequence with $2 \times M_U \times N_U$ computation statements ($M_U \times N_U$ if $FMA = 1$).

2. The $M_U + N_U$ loads of elements of A and B are injected into the resulting computation sequence by scheduling a block of I_F loads in the beginning and blocks of N_F loads thereafter as needed.

3. The K_U iterations of the k'' loop are completely unrolled.

4. The k' loop is software-pipelined so that operations from the current iteration are overlapped with operations from the previous iteration.

Table 1 lists all optimization parameters for future reference.

Name	Description
N_B	L1 data cache tile size
M_U, N_U	Register tile size
K_U	Unroll factor for k' loop
L_s	Latency for computation scheduling
FMA	1 if fused multiply-add should be assumed, 0 otherwise
F_F, I_F, N_F	Scheduling of loads

Table 1: Summary of optimization parameters

Finally, ATLAS copies portions 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 4. ATLAS also incorporates a simple form of tiling for the L2 cache, called CacheEdge; we will not discuss this because our focus in the mini-MMM code, which is independent of CacheEdge.

2.2 Global search in ATLAS

It is intuitively obvious that the performance of the generated mini-MMM code suffers if the values of the optimization 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.

To find optimal values for the optimization parameters, ATLAS uses a global search strategy called *orthogonal line search* [11]. This is a general optimization strategy that tries to find the optimal value of a function $y = f(x_1, x_2, ..., x_n)$ by reducing the n-dimensional optimization problem into a sequence of n 1-dimensional optimization problems by ordering the parameters x_i in some order, and optimizing them one at a time in that order, using reference values for parameters that have not been optimized yet. 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 if the parameters $x_1, x_2, ..., x_n$ are more or less independent. The specific order used in ATLAS is: N_B; (M_U N_U); K_U; L_s F_F, I_F, and N_F. Details can be found in [18].

2.3 Model-driven optimization

We now describe a model for estimating values for optimization parameters [17]. This model is used to generate mini-MMM code using the ATLAS code generator, as shown in Figure 5.

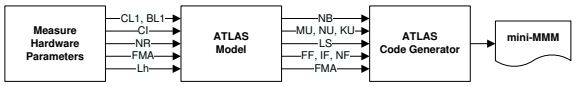


Figure 5: Model-driven Optimization Architecture

This model requires the following machine parameters.

- C_{L1}, B_{L1} – capacity and line size of the L1 data cache
- C_I – capacity of the instruction cache
- N_R – number of floating-point registers
- L_s – as measured by the ATLAS micro-benchmark
- FMA – existence of a fused multiply-add instruction

Figure 6 describes the model. The rationale for the model is as follows.

The simplest model for N_B is to choose its value so that all three blocks of matrices A, B, and C can reside in the cache simultaneously. This gives the following inequality.

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

A more careful analysis shows that when multiplying two matrices, capacity misses can be avoided completely if one of the matrices, a row or a column of another matrix, and an element of the third matrix can be cache resident simultaneously [17]. This analysis assumes that the cache replacement policy is optimal. It yields the following inequality for N_B.

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

Finally, we must correct for non-unit cache line size (B_{L1}) and LRU replacement policy, which yields the model shown in Figure 6 [17].

- Choose largest N_B, which satisfies:

$$\left\lceil \frac{N_B^2}{B_{L1}} \right\rceil + 3 \left\lceil \frac{N_B}{B_{L1}} \right\rceil + 1 \leq \frac{C_{L1}}{B_{L1}}. \tag{1}$$

- Choose M_U and N_U as follows:

$$N_U \leftarrow \left\lfloor \sqrt{N_R - L_s + 1} - 1 \right\rfloor$$
$$M_U \leftarrow \frac{N_R - L_s - N_U}{N_U + 1}.$$

- Use L_s as determined by the ATLAS micro-benchmark.
- Use FMA as determined by the ATLAS micro-benchmark.
- Set $F_F = 1$ and $I_F = N_F = 2$.

Figure 6: Model for estimating optimal values of optimization parameters

The micro-MMM produced by the ATLAS code generator requires $M_U \times N_U$ registers for storing the elements of C, M_U registers to store elements of A, N_U registers for storing the elements of B, and L_s registers to store the temporary results of multiplies, which will be consumed by dependent additions. Therefore it is necessary that

$$M_U \times N_U + M_U + N_U + L_s \leq N_R. \quad (4)$$

To obtain the model shown in Figure 6, we solve Inequality(4) assuming $N_U = M_U$ and set N_U to its largest integer solution. We then compute M_U as the largest integer solution of Inequality(4), based on the computed value for N_U.

Finally, we found that performance was relatively insensitive to the values of F_F, I_F and N_F, perhaps because the back-end compilers themselves rescheduled operations, so we fixed their values as shown in Figure 6.

2.4 Discussion

It should be noted that similar optimization parameters are needed even if one uses so-called *cache-oblivious* algorithms [9]. For example, when implementing a cache-oblivious matrix multiplication code, one needs to switch from recursive to iterative code when the problem size becomes small enough to fit in the L1 cache. This requires a parameter similar to N_B [3, 5].

3. ANALYZING THE GAP

We compared the performance of code generated by ATLAS and by the model-driven version of ATLAS on ten different architectures. In this section, we discuss only the machines where there are interesting performance differences between the code generated by the original ATLAS system and by the model-driven version we built.

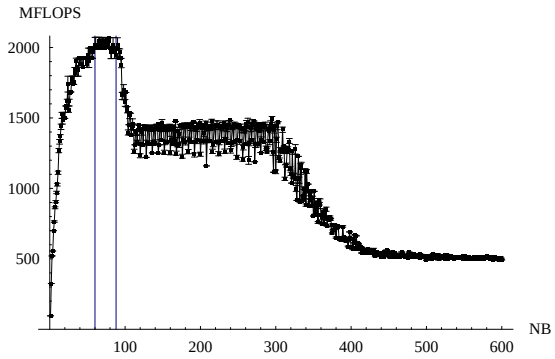
3.1 AMD Opteron 240

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 (compare "Model" and "Global Search" in Figure 7(d)). To understand the problem, we studied the optimization parameter values produced by the two approaches. These values are shown in Table 12(a) (to permit easy comparisons between the different approaches explored in this paper, the parameter values used by all approaches are shown together in that section). Although the two sets of values are quite different, this by itself is not necessarily significant. For example, Figure 7(b) shows how performance of the mini-MMM code changes as a function of N_B, while 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.

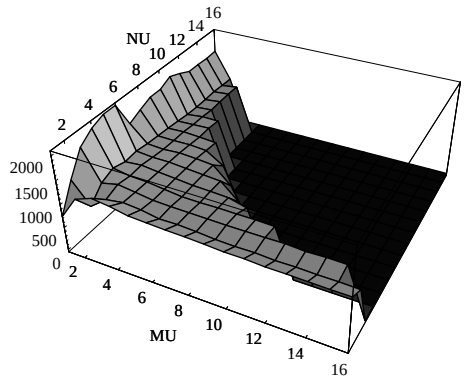
On the other hand, performance sensitivity to M_U and N_U, shown in Figure 7(c), demonstrates that the optimal values of (M_U, N_U) are $(6, 1)$. Notice that this graph is not symmetric with respect to M_U and N_U, because an $M_U \times 1$ tile of C is contiguous in memory, but a $1 \times N_U$ tile is not [15]. 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.

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

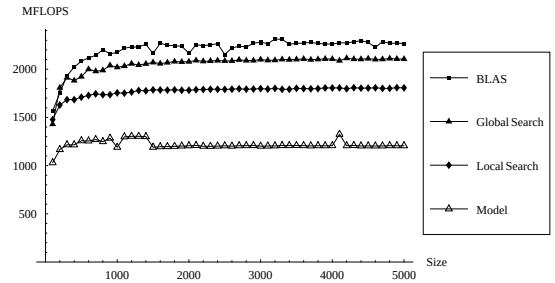
(a) Platform Specification



(b) Sensitivity to N_B



(c) Sensitivity to M_U and N_U



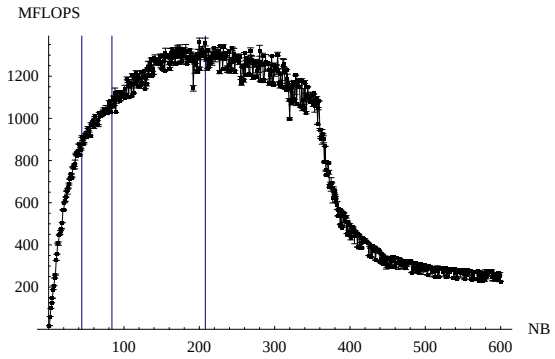
(d) MMM Results

Figure 7: AMD Opteron 240

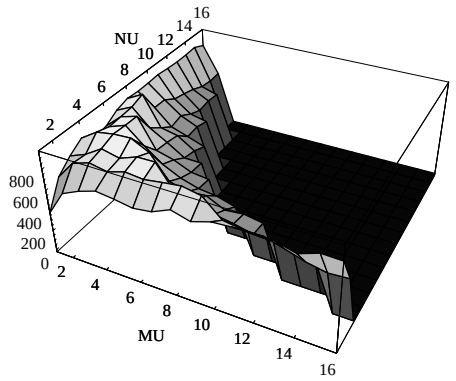
Since the difference in performance between the code produced by global search and the code produced by using the model is about 40%, it is likely that there is a problem with the model presented in Section 2.3 for determining (M_U, N_U). Evidence for this is provided by the line "Local Search" in Figure 7(d), which shows that there is significant performance gap even if we perform local search, described in more detail in Section 4.2, around the parameter values estimated by the model. In Section 4.1.1, we show how model refinement fixes this problem.

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

(a) Platform Specification



(b) Sensitivity to N_B



(c) Sensitivity to M_U and N_U

Figure 8: SUN UltraSPARC IIIi

3.2 SUN UltraSPARC IIIi

The optimization parameters derived by using the model and global search are shown in Table 13(a). Figure 13(c) 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 expose a deficiency of the model that afflicts ATLAS Global Search as well. The problem lies in the choice of N_B. Figure 8(b) 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.

At first sight, it is surprising that the optimal value of N_B is so much larger than the model-predicted value. However, when we studied the sensitivity results closely, we realized that these results could be explained by the fact that on this machine, it is actually more beneficial to tile for the L2 cache rather than the L1 cache. Notice that performance increases with increasing values of N_B up to $N_B \approx 210$. Performance degrades for $N_B > 210$, because three tiles do not fit together in the L2 cache anymore ($3 \times N_B^2 > C_{L2}$), and L2 conflict misses start to occur. At $N_B \approx 360$, there is a second, more pronounced, performance drop, which can be explained by applying Inequality (1) for the L2 cache. After this point, L2 capacity misses start to occur. Notice that there are no drops in performance around $N_B = 52$ and $N_B = 88$ which are the corresponding values for the L1 data cache.

In general, it is desirable to tile for the L2 cache 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. Of course, another possibility is to do multi-level cache tiling but the ATLAS code generator permits tiling for only a single level of the cache hierarchy. In Section 4.1.2, we show how model refinement and local search solve this problem.

We also noticed that even if we tile for the best cache level, the model sometimes computes an N_B value which is slightly larger than the optimum. Our study revealed that this effect arose because we ignored the interaction of register tiling and cache tiling. In particular, the extra level of tiling for the register file changes the order of the scalar operations inside the mini-MMM. Therefore, when computing the optimal N_B, one should consider horizontal panels of width M_U and vertical panels of width N_U instead of rows and columns from matrix A and matrix B respectively. In Section 4.1.2, we show how model refinement can take this interaction into account.

3.3 Intel Itanium 2

Figure 9(b) 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 with the SUN UltraSPARC IIIi, tiling for the L1 data cache is not beneficial (in fact,we found out that the Itanium does not cache floating-point values in the L1 cache). On this platform, even the L2 cache is "invisible", and the drops in performance are explained by computing the blocking parameters with respect to the L3 cache whose capacity is $C_{L3} = 3MB$.

Figure 9(c) 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 $N_B = 362$ computed by $3 \times N_B \leq C_{L3}$ is not nearly as good as $N_B = 360$. 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 cache tile boundary. The number of conflict misses in the L1 and L2 caches can also vary with tile size. It is difficult to model conflict misses analytically. Therefore, to address these problems, we advocate using local search around the model-predicted value, as discussed in detail in Section 4.2.1.

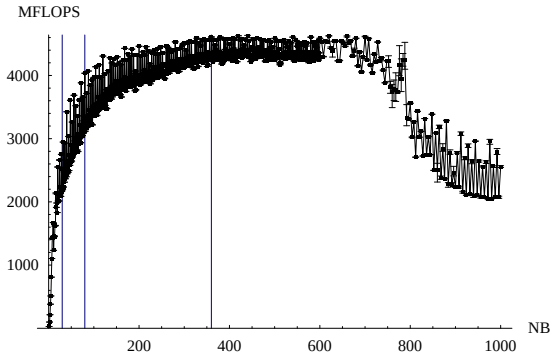
For completeness, we show the sensitivity of performance to M_U and N_U in Figure 9(d), although there is nothing interesting in this graph. Because of the extremely large number of registers on this platform ($N_R = 128$), the peak of the hill is more like a plateau with a multitude of good choices for the M_U and N_U unroll factors. Both Model and Global Search do well.

3.4 Summary

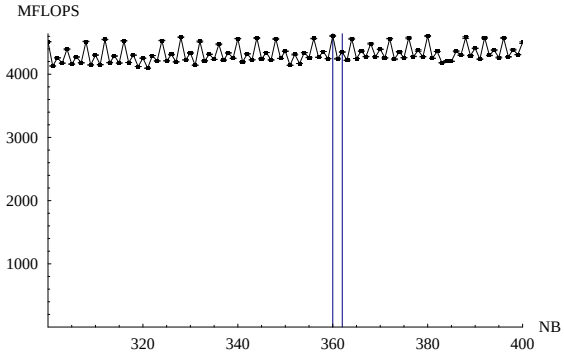
Our experiments point to a number of deficiencies with the

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

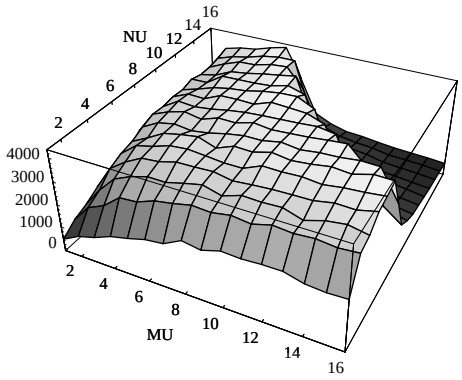
(a) Platform Specification



(b) Sensitivity to N_B



(c) Sensitivity to N_B(zoomed)



(d) Sensitivity to M_U and N_U

Figure 9: Intel Itanium 2

model in [17]. 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. The model for N_B does not consider interactions with register-tiling. Finally, conflict misses can reduce the effective value of N_B on some machines.

4. CLOSING THE GAP

We now show how the performance gaps discussed in Section 3 can be eliminated by a combination of model refinement and local search. We discuss model refinement in Section 4.1, and local search in Section 4.2.

4.1 Model refinement

4.1.1 *Refining model for M_U and N_U*

Recall that Inequality (4) for estimating values for M_U and N_U, reproduced below for convenience, is based on an allocation of registers for a $M_U \times N_U$ sub-matrix ($\bar{c}$) of C, a $M_U \times 1$ vector ($\bar{a}$) of A, a $1 \times N_U$ vector ($\bar{b}$) of B, and L_s temporary values.

$$M_U \times N_U + M_U + N_U + L_s \leq N_R$$

Allocating the complete vectors $\bar{a}$ and $\bar{b}$ in registers is justified by the reuse of their elements in computing the outer product $\bar{c} = \bar{c} + \bar{a} \times \bar{b}$. During this procedure each element of $\bar{a}$ is accessed N_U times and each element of $\bar{b}$ is accessed M_U times.

However, these arguments implicitly make the following assumptions.

1. $M_U > 1$ and $N_U > 1$: if $M_U = 1$ each element of $\bar{b}$ is accessed only one time and therefore there is not enough reuse to justify allocating $\bar{b}$ to registers. By analogy the same holds for N_U and $\bar{a}$ respectively.

2. The ISA is three-address: it is assumed that an element of $\bar{a}$ and an element of $\bar{b}$ can be multiplied and the result stored in a third location, without overwriting any of the input operands, as they need to be reused in other multiplications. This is impossible on a two-address code ISA.

Both these assumptions are violated on the AMD Opteron. The Opteron ISA is based on the x86 ISA and is therefore two-address. Moreover, the Opteron has only 8 registers and no fused multiply-add instruction, so the model in Figure 6 estimates that the optimal value of N_U is 1. Therefore, there is no reuse of elements of $\bar{a}$. Not surprisingly, the original model does not perform well on this architecture.

In fact, Table 12(a) shows that the ATLAS global search chose $(M_U, N_U) = (6, 1)$, which provides the best performance as can be seen in Figure 7(c), although it violates Inequality (4). Additionally, global search chose $FMA = 1$ even though there is no fused multiply-add instruction!

To understand why code produced with these parameters achieves high performance, we examined the machine code output by the C compiler. A stylized version of this code is shown in Figure 10. It uses 1 register for $\bar{b}$ (rb), 6 registers for $\bar{c}$ ($rc_1 \ldots rc_6$) and 1 temporary register (rt) to hold elements of $\bar{a}$.

One might expect that this code will not perform well because there are dependencies between back-to-back adjacent instructions and a single temporary register rt is used. The reason why this code performs well in practice is because the Opteron is an *out-of-order issue* architecture with *register renaming*. Therefore it is possible to schedule several multiplications in successive cycles without waiting for the first one to complete, and the the single temporary register rt is renamed to a different physical register for each pair of multiply-add instructions.

```
rc1 ← c̄1 ... rc6 ← c̄6
...
loop k
{
    rb ← b̄1

    rt ← ā1
    rt ← rt × rb
    rc1 ← rc1 + rt

    rt ← ā2
    rt ← rt × rb
    rc2 ← rc2 + rt

    ⋮

    rt ← ā6
    rt ← rt × rb
    rc6 ← rc6 + rt
}
...
c̄1 ← rc1 ... c̄6 ← rc6
```

Figure 10: $(M_U, N_U) = (6, 1)$ code for x86 CISC

To verify these conclusions, we used a different mode of floating-point operations on the Opteron in which it uses the 16 SSE vector registers to hold scalar values. In this case, the model predicts $(M_U, N_U) = (3, 3)$. However, the ISA is still two-address code and experiments show that better performance can be achieved by $(M_U, N_U) = (14, 1)$ [15].

We conclude that when an architecture has a small number of registers or two-address code ISA, but implements out-of-order issue with register renaming, it is better to leave instruction scheduling to the processor and use the available registers to allocate a larger register tile. These insights permit us to refine the original model for such architectures as follows: $N_U = 1$, $M_U = N_R - 2$, $FMA = 1$.

To finish this story, it is interesting to analyze how the ATLAS search engine settled on these parameter values. Note that on a processor that does not have a fused multiply-add instruction, $FMA = 1$ is equivalent to $FMA = 0$ and $L_s = 1$. The code produced by the ATLAS Code Generator is shown schematically in Figure 11. Note that this code uses 6 registers for $\bar{a}$ ($ra_1 \ldots ra_6$), 1 register for $\bar{b}$ (rb), 6 registers for $\bar{c}$ ($rc_1 \ldots rc_6$) and 1 temporary register (implicitly by the multiply-add statement). However, the back-end compiler (GCC) reorganizes this code into the code pattern shown in Figure 10.

```
rc1 ← c̄1 ... rc6 ← c̄6
...
loop k
{
    ra1 ← ā1
    rb ← b̄1
    rc1 ← rc1 + ra1 × rb
    ra2 ← ā2
    ra3 ← ā3
    rc2 ← rc2 + ra2 × rb
    rc3 ← rc3 + ra3 × rb
    ra4 ← ā4
    ra5 ← ā5
    rc4 ← rc4 + ra4 × rb
    rc5 ← rc5 + ra5 × rb
    ra6 ← ā6
    rc6 ← rc6 + ra6 × rb
}
...
c̄1 ← rc1 ... c̄6 ← rc6
```

Figure 11: ATLAS code for $(M_U, N_U) = (6, 1)$

Notice that the ATLAS Code Generator itself is not aware that the code of Figure 10 is optimal. However, setting $FMA = 1$ (even though there is no fused-multiply instruction) produces code that triggers the right instruction reorganization heuristics inside GCC, and performs well on the Opteron. This illustrates the well-known point that search does not need to be intelligent to do the right thing! Nevertheless, our refined model explains the observed performance data, makes intuitive sense, and can be easily incorporated into a compiler.

Although there is a large body of existing work on register allocation and instruction scheduling for pipelined machines [7, 10], we are not aware of prior work that has highlighted this peculiar interaction between compile-time scheduling, register allocation, dynamic register-renaming, and out-of-order execution.

4.1.2 Refining N_B

We made a refinement to the model for N_B to correct for the interaction between register tiling and cache tiling as follows. When computing the optimal N_B, we consider horizontal panels of width M_U and vertical panels of width N_U instead of rows and columns from matrix A and matrix B respectively. This leads to the following Inequality (5).

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B \times N_U}{B_1} \right\rceil + \left\lceil \frac{M_U}{B_1} \right\rceil \times N_U \leq \frac{C_1}{B_1} \qquad (5)$$

Finally, to avoid fractional register tiles, we trim the value of N_B obtained from Inequality (5) so that it is a multiple of M_U and N_U.

We now describe how we tile for the right cache level. The models presented in Section 2.3 do 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 Inequality (5), 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.

4.1.3 Refining L_s

L_s is the optimization parameter that represents the skew factor the ATLAS Code Generator uses when scheduling dependent multiplication and addition operations for the CPU pipeline. Although the ATLAS documentation incorrectly states that L_s is the length of the floating point pipeline [16], the value is determined empirically in a problem dependent way, which allows global search (and the original model) to achieve good performance. In this section we present our

analytical model for L_s, which relies only on hardware parameters.

Studying the description of the scheduling in Section 2.1, we see that the schedule effectively executes L_s independent multiplications and $L_s - 1$ independent additions between a multiplication mul_i and the corresponding addition add_i. The hope is that these $2L_s - 1$ independent instructions will hide the latency of the multiplication. If the floating-point units are fully pipelined and the latency of multiplication is L_h, we get the following inequality, which can be solved to obtain a value for L_s.

$$2L_s - 1 \geq L_h \tag{6}$$

On some machines, there are multiple floating-point units. If $|ALU_{FP}|$ is the number of floating-point ALUs, Inequality (6) gets refined as follows.

$$\frac{2L_s - 1}{|ALU_{FP}|} \geq L_h \tag{7}$$

Solving Inequality (7) for L_s, we obtain Inequality (8).

$$L_s = \left\lceil \frac{L_h \times |ALU_{FP}| + 1}{2} \right\rceil \tag{8}$$

Of the machines in our study, only the Intel Pentium machines have floating-point units that are not fully pipelined; in particular, multiplications can be issued only once every 2 cycles. Nevertheless, this does not introduce any error in our model because ATLAS does not schedule back-to-back multiply instructions, but intermixes them with additions. Therefore, Inequality (6) holds.

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_{B_M} is the value of N_B estimated by the model, we can refine this value by local search in the interval $[N_{B_M} - lcm(M_U, N_U), N_{B_M} + lcm(M_U, N_U)]$. This ensures that we examine the first values of N_B in the neighborhood of N_{B_M} 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

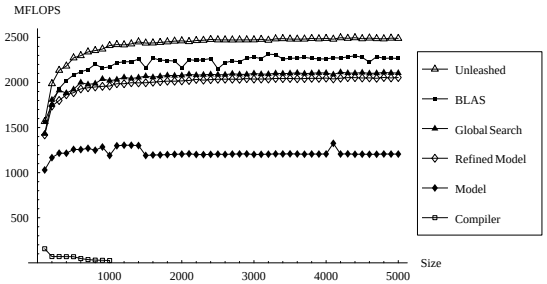
We evaluated the following six approaches on a large number of modern plarforms, including DEC Alpha 21264, IBM

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
Model	88	2, 1, 88	2	0	0, 2, 2	1189
Refined Model	88	6, 1, 88	1	1	0, 2, 2	2050
Local Search	88	6, 1, 88	1	1	0, 2, 2	2050
ML Local Search	88	6, 1, 88	1	1	0, 2, 2	2050
Global Search	60	6, 1, 60	6	1	0, 6, 1	2072
Unleashed	56					2608

(a) Optimization Parameters

	Parameters		Total
	Machine	Optimization	(sec)
Model	101	2	103
Refined Model	101	2	103
Local Search	101	31	132
ML Local Search	101	126	227
Global Search	148	375	523

(b) Timings



(c) MMM Results

Figure 12: AMD Opteron 240

Power 3/4, SGI R12000, SUN UltraSPARC IIIi, Intel Pentium III/4, Intel Itanium 2, AMD Athlon MP, and AMD Opteron 240. For lack of space, we describe results only for those machines discussed earlier. We used ATLAS v.3.6.0, which is the latest stable version of ATLAS as of this writing.

1. Model: This approach uses the model of Yotov et al. [17] as described in Section 2.3.
2. Refined Model: This approach uses the refined models described in Section 4.1.
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 by accepting all defaults that the ATLAS team have provided. Although this is not directly relevant to our study, we include it anyway for completeness.

Global Search uses the micro-benchmarks in the standard ATLAS distribution to determine the necessary machine parameters. For all other approaches we used X-Ray [19], which is a tool implemented by us for accurately measuring machine parameters.

5.1 AMD Opteron 240

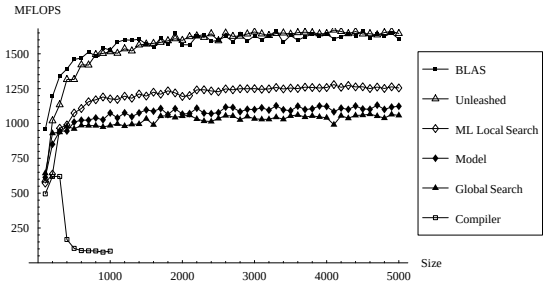
Figure 12 shows the results for AMD Opteron 240. For native BLAS we used ACML 2.0. The MMM performance achieved by Model + Local Search is only marginally worse than that of Global Search. Additional analysis showed that

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
Model	84	4, 4, 84	4	0	0, 2, 2	1120
Refined Model	84	4, 4, 84	4	0	0, 2, 2	1120
Local Search	84	4, 4, 84	4	0	0, 2, 2	1120
ML Local Search	208	4, 4, 16	4	0	0, 2, 2	1308
Global Search	44	4, 3, 44	5	0	0, 3, 2	986
Unleashed	168					1694

(a) Optimization Parameters

	Parameters		Total
	Machine	Optimization	(sec)
Model	112	7	119
Refined Model	112	7	119
Local Search	112	127	239
ML Local Search	112	496	608
Global Search	203	1233	1436

(b) Timings



(c) MMM Results

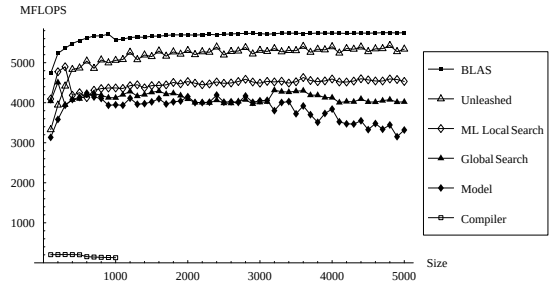
Figure 13: SUN UltraSPARC IIIi

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
Model	30	10, 10, 4	1	1	0, 2, 2	3130
Refined Model	30	10, 10, 4	1	1	0, 2, 2	3130
Local Search	30	10, 10, 4	1	1	0, 2, 2	3130
ML Local Search	360	10, 10, 4	1	1	0, 2, 2	4602
Global Search	80	10, 10, 4	4	1	0, 19, 1	4027
Unleashed	120					4890

(a) Optimization Parameters

	Parameters		Total
	Machine	Optimization	(sec)
Model	143	6	149
Refined Model	143	6	149
Local Search	143	162	305
ML Local Search	143	278	421
Global Search	1554	29667	31221

(b) Timings



(c) MMM Results

Figure 14: Intel Itanium 2

this 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 found the optimal value.

We conclude that the model refinement and local search described in Section 4.1 are sufficient to address the performance problems with the basic model described in Section 3.

5.2 SUN UltraSPARC IIIi

Figure 13 shows the results for SUN UltraSPARC IIIi. We used the native BLAS library included in Sun One Studio 9.0. 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 restricted the N_B interval examined by Global Search, leading to a suboptimal value of $N_B$and lower 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.

5.3 Intel Itanium 2

Figure 14 shows the results for Intel Itanium 2. Native BLAS used is MKL 6.1. Model does not perform well because it tiles for the L1 data cache. ATLAS global search selected the maximum value in its 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

Finally, we include some interesting results for the SGI R12K. Figure 15 shows the results for this machine. For native BLAS we used SGI SCSL v.1.4.1.3. 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

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

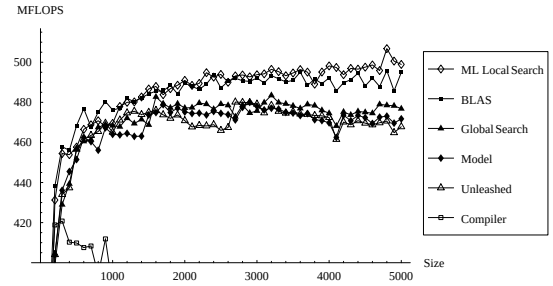
(a) Platform Specification

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
Model	58	5, 4, 58	1	1	0, 2, 2	440
Refined Model	58	5, 4, 58	1	1	0, 2, 2	440
Local Search	58	5, 4, 58	1	1	0, 2, 2	440
ML Local Search	418	5, 4, 16	1	1	0, 2, 2	508
Global Search	64	4, 5, 64	1	0	1, 8, 1	457
Unleashed	64					463

(b) Optimization Parameters

	Parameters		Total
	Machine	Optimization	(sec)
Model	118	13	131
Refined Model	118	13	131
Local Search	118	457	575
ML Local Search	118	496	608
Global Search	251	2131	2382

(c) Timings



(d) MMM Results

Figure 15: SGI R12000

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. In previous work, we have shown that model-driven optimization can produce BLAS codes with performance within 10-20% of that of code produced by empirical optimization. In this paper, we have shown that this remaining performance gap can be eliminated by a combination of model refinement and local search, without increasing search time substantially. The model refinement (i) corrected the computation of the register tile parameters (M_U, N_U) for machines which have either a 2-address ISA or relatively few logical registers, and (ii) adjusted the value of the cache tile parameter (N_B) to take interactions with register tiles into account. The local search allowed us to take L1 conflict misses into account, and to open up the possibility of tiling for lower levels of the memory hierarchy. On some machines, this strategy outperformed ATLAS Global Search and the native BLAS.

We believe that this combination of model refinement and local search is promising, and it is the corner-stone of a system we are building for generating dense numerical linear algebra libraries that are optimized for many levels of the memory hierarchy, a problem for which global search is not tractable. We also believe that this approach is the most promising one for incorporation into general-purpose compilers.

7. REFERENCES

[1] Automatically Tuned Linear Algebra Software (ATLAS). `http://math-atlas.sourceforge.net/`.

[2] R. Allan and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.

[3] Gianfranco Bilardi, Paolo D'Alberto, and Alex Nicolau. Fractal matrix multiplication: A case study on portability of cache performance. In *Algorithm Engineering: 5th International Workshop, WAE*, 2001.

[4] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 279–290, 1995.

[5] Paolo D'Alberto and Alex Nicolau. Juliusc: A practical approach for the analysis of divide-and-conquer algorithms. In *LCPC*, 2004.

[6] Jack Dongarra. Personal communication.

[7] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Register pipelining: An integrated approach to register allocation for scalar and subscripted variables. In *Proceedings of the 4th International Conference on Compiler Construction*, pages 192–206. Springer-Verlag, 1992.

[8] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[9] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 285. IEEE Computer Society, 1999.

[10] Daniel M. Lavery, Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. The importance of prepass code scheduling for superscalar and superpipelined processors. *IEEE Trans. Comput.*, 44(3):353–370, 1995.

[11] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C*. Cambridge University Press, 2002.

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

[13] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect of benchmark run. Technical Report CSD-93-767, February 1993.

[14] Robert Schreiber and Jack Dongarra. Automatic blocking of nested loops. Technical Report CS-90-108, Knoxville, TN 37996, USA, 1990.

[15] R. Clint Whaley. `http://sourceforge.net/mailarchive/forum.php?thread_id=1569256&forum_id=426`.

[16] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[17] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 63–76. ACM Press, 2003.

[18] Kamen Yotov, Xiaoming Li, Gang Ren, Maria Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[19] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. In *Proc. of the 2005 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*.