INVITED PAPER

An Experimental Study of Self-Optimizing Dense Linear Algebra Software

Analytical models of the memory hierarchy are used to explain the performance of self-optimizing software.

By Milind Kulkarni and Keshav Pingali, *Member IEEE*

ABSTRACT | Memory hierarchy optimizations have been studied by researchers in many areas including compilers, numerical linear algebra, and theoretical computer science. However, the approaches taken by these communities are very different. The compiler community has invested considerable effort in inventing loop transformations like loop permutation and tiling, and in the development of simple analytical models to determine the values of numerical parameters such as tile sizes required by these transformations. Although the performance of compiler-generated code has improved steadily over the years, it is difficult to retarget restructuring compilers to new platforms because of the need to develop analytical models manually for new platforms. The search for performance portability has led to the development of *self-optimizing software systems*. One approach to self-optimizing software is the *generate-and-test* approach, which has been used by the dense numerical linear algebra community to produce high-performance BLAS and fast Fourier transform libraries. Another approach to portable memory hierarchy optimization is to use the divide-and-conquer approach to implementing *cache-oblivious* algorithms. Each step of divide-and-conquer generates problems of smaller size. When the working set of the subproblems fits in some level of the memory hierarchy, that subproblem can be executed without capacity misses at that level. Although all three approaches have been studied extensively, there are few experimental studies that have compared these approaches. How well does the code produced by current self-optimizing systems perform compared to hand-tuned code? Is empirical search essential to the generate-and-test approach or is it possible to use analytical models with platform-specific parameters to reduce the size of the search space? The cache-oblivious approach uses divide-and-conquer to perform approximate blocking; how well does approximate blocking perform compared to precise blocking? This paper addresses such questions for matrix multiplication, which is the most important dense linear algebra kernel.

KEYWORDS | Algorithms; cache blocking; cache memories; computer performance; linear algebra; matrix multiplication; memory architecture; tiling

Manuscript received May 16, 2007; revised December 3, 2007. The work of M. Kulkarni was supported by the U.S. Department of Energy under an HPCS Fellowship. The work of K. Pingali was supported by the National Science Foundation under Grants 0719966, 0702353, 0615240, 0541193, 0509307, 0509324, 0426787, and 0406380; by IBM Corporation; and by Intel Corporation.

The authors are with the Department of Computer Science, The University of Texas at Austin, Austin, TX 78712-0233 USA (e-mail: milind@cs.utexas.edu; pingali@cs.utexas.edu).

Digital Object Identifier: 10.1109/JPROC.2008.917732

I. INTRODUCTION

The performance of most programs on modern computers is limited by the performance of the memory system. Since the latency of memory accesses can be many hundreds of cycles, the processor may be stalled most of the time, waiting for loads to complete; moreover, the bandwidth from memory is usually far less than the rate at which the processor can consume data.

Both problems can be addressed by using caches—if most memory requests are satisfied by some cache level, the effective memory latency as well as the bandwidth required from memory are reduced. As is well known, the effectiveness of caching for a problem depends both on the algorithm used to solve the problem and on the program used to express that algorithm (simply put, an algorithm defines only the dataflow of the computation, while a program for a given algorithm also specifies the schedule of operations and may perform storage allocation consistent with that schedule). One useful quantity in thinking about these issues is *algorithmic data reuse*, which is an abstract measure of the number of accesses made to a typical memory location by the algorithm. For

0018-9219/$25.00 ©2008 IEEE

example, the standard algorithm for multiplying matrices of size $n \times n$ performs $O(n^3)$ operations on $O(n^2)$ data, so it has excellent algorithmic data reuse since each data element is accessed $O(n)$ times; in contrast, matrix transpose performs $O(n^2)$ operations on $O(n^2)$ data, so it has poor algorithmic data reuse. When an algorithm has substantial algorithmic data reuse, the challenge is to write the program so that the memory accesses made by that program exhibit both spatial and temporal locality. In contrast, programs that encode algorithms with poor algorithmic data reuse must be written so that spatial locality is exploited.

As is well known, it is tedious to optimize programs by hand for memory hierarchies; for example, the standard algorithm for matrix multiplication can be coded as a single assignment statement within three nested loops, but an optimized version within a typical basic linear algebra subprogram (BLAS) library can be thousands of lines long.

One approach to reduce the tedium of manual optimization is to use restructuring compiler technology to transform high-level programs into lower level programs that are optimized for a particular machine and memory hierarchy [1]–[3]. For dense linear algebra programs, the most important transformations are loop tiling [4]–[6] and loop interchange [4]; linear loop transformations like skewing and reversal may be performed first to enable these transformations [7]–[10]. Some transformations such as loop tiling require numerical parameters whose optimal values may depend on the parameters of the memory hierarchy such as cache capacities. Currently, compilers use simple architectural models to determine these values. The quality of compiler-generated code has improved steadily over the years; for example, the matrix multiplication code produced by the Intel compiler for the Itanium 2, starting from the three nested-loop version, runs at roughly 92% of peak, whereas the handwritten Goto BLAS code runs at almost 99% of peak.[1]

Although restructuring compilers are reaching maturity, they are big, complicated programs, and it is not easy to retarget them to new platforms. The quest for performance portability has led to the development of *self-optimizing software systems* that can automatically tune themselves to different platforms without manual intervention.

One general approach to building self-optimizing systems is *generate-and-test:* a program generator is used to generate a large number of implementations of one or more algorithms for some problem, and the best implementation for a given machine is determined by running all these implementations (programs) on that machine using sample input data. This approach is sometimes called *empirical search* since the system performs a search over a large space of different implementations, using empirical measurements to evaluate the performance of each point in the search space. The ATLAS system,[2] which generates BLAS libraries, and FFTW [11], which generates fast Fourier transform (FFT) libraries, are well-known implementations of the generate-and-test approach.

For memory hierarchies, a different self-optimization strategy is to use recursive, divide-and-conquer algorithms. For example, to multiply two matrices A and B, we can divide one of the matrices (say, A) into two submatrices A_1 and A_2 and multiply A_1 and A_2 by B; the base case of this recursion is reached when both A and B have a single element. Programs written in this divide-and-conquer style perform *approximate* blocking in the following sense. Each division step generates subproblems of smaller size. When the working set of some subproblem fits in a given level of cache, the computation can take place without suffering capacity misses at that level. The resulting blocking is only approximate since the size of the working set may be smaller than the capacity of the cache.

An important theoretical result about divide-and-conquer algorithms was obtained in 1981 by Hong and Kung [12], who also introduced the *I/O model* to study the memory hierarchy performance of algorithms. This model considers a two-level memory hierarchy consisting of a cache and main memory. The processor can compute only with values in the cache, so it is necessary to move data between cache and memory during program execution. The *I/O complexity* of a program is an asymptotic measure of the total volume of data movement when that program is executed. Hong and Kung showed that divide-and-conquer programs for matrix multiplication and FFT are optimal under this measure. In 1997, Gustavson implemented highly optimized recursive versions of common linear algebra routines [13]. In 1999, Frigo *et al.* extended the Hong and Kung result to matrix transposition; they also coined the adjective *cache-oblivious* to describe these programs because they self-adapt to memory hierarchies even though cache parameters are not reflected in the code [14].

There are a number of questions that one can ask about self-optimizing systems. How well does the code produced by self-optimizing systems perform compared to hand-optimized code? Is empirical search essential to the generate-and-test approach or is it possible to use simple analytical models to reduce the size of the search space, perhaps down to a single point? The cache-oblivious approach uses divide-and-conquer to perform approximate blocking; how close to optimal do block sizes have to be to ensure good performance?

The goal of this paper is to address some of these questions in the context of dense linear algebra, using the Itanium 2 as an experimental platform. There are several reasons why this is a good domain for such a study. First, there are publicly available, highly tuned, hand-optimized codes to serve as a basis for evaluating the performance of

[1]http://www.cs.utexas.edu/users/flame/goto/.

[2]http://math-atlas.sourceforge.net/.

code produced by self-optimizing approaches. Secondly, it is possible to design simple analytical models for this domain to understand performance abstractly. Lastly, linear algebra is an important area with applications in many domains ranging from scientific computing to Internet search.

The rest of this paper is organized as follows. In Section II, we describe a novel analytical approach for understanding the performance of matrix multiplication on a memory hierarchy. This approach leads naturally to considerations of loop interchange and loop tiling for improving the performance of matrix multiplication. We present experimental results that show that the Intel compiler for the Itanium, which implements these and other transformations, can generate high-quality code with near peak performance. We then shift our focus to self-optimizing systems. In Section III, we describe the ATLAS system for generating optimized BLAS libraries. ATLAS uses the generate-and-test approach to determine near-optimal values for certain parameters used by its code generator. To study the importance of search, we replace this empirical search with a simple analytical model for computing these values directly and show that the performance of the resulting code is close to that of code produced by the ATLAS system. These results suggest that the search space can be reduced dramatically without compromising on the quality of the produced code. In Section IV, we describe the divide-and-conquer approach to memory hierarchy optimization. We motivate approximate blocking by giving a quantitative analysis of how blocking can reduce the required bandwidth from memory. This analysis provides a novel way of thinking about the I/O optimality of recursive algorithms for problems like matrix multiplication. We then describe experiments that show that cache-oblivious programs may not perform as well as hand-tuned cache-conscious programs and suggest how this performance can be improved.[3]

II. ANALYTICAL PERFORMANCE MODELS

In this section, we develop simple analytical models to explain important aspects of the memory hierarchy performance of matrix multiplication. In addition to providing insight, these models can be useful for reducing search time in generate-and-test approaches, as we discuss in Section III.

Fig. 1 shows the standard three nested-loop version of matrix multiplication. As is well known, this loop nest is *fully permutable* because the loops can be executed in any order without changing the result [3]. To refer to these versions, we will specify the loop order from outermost in; for example, the *ijk* version is the one with the *i* loop

[3]Some of the experimental results discussed in this paper were presented in earlier papers from our group [15], [16].

```
for i ∈ [0 : 1 : N − 1]
    for j ∈ [0 : 1 : M − 1]
        for k ∈ [0 : 1 : K − 1]
            C_ij = C_ij + A_ik * B_kj
```

Fig. 1. *Naïve MMM code.*

outermost and the *k* loop innermost. Without loss of generality, we will assume that the matrix is stored in row-major order.

Fig. 2 shows the L2 cache miss ratio for these matrix–matrix multiplication (MMM) versions on an Itanium 2. The capacity of the L2 cache is 256 KB and its line size is 128 bytes. We focused on the L2 cache because floating-point loads and stores bypass the L1 cache on the Itanium architecture. All three matrices are square, and the matrix elements are 8-byte doubles. We used the Performance Application Programming Interface (PAPI) toolkit[4] to measure L2 data cache accesses and misses.[5]

Careful examination of Fig. 2 shows that for each loop order, the range of matrix sizes can be divided into three zones with characteristic behavior. For small matrix sizes, we see that the miss ratio *decreases* as the matrix size increases, which is counterintuitive (Fig. 3 zooms in on smaller matrix sizes, making this behavior more clear). After reaching a minimum, the miss rates begin to increase until they stabilize; for medium matrix sizes, the miss ratio is roughly constant irrespective of problem size. Then, as the matrix size increases further, the miss ratio increases again until it finally levels off. However, the point at which the miss rate begins to increase, as well as the asymptotic miss ratio, depend on the loop order.

A. Miss Ratio Calculations

Although Fig. 2 is very complex, analytical models can be used to explain the more important features. If we assume that matrix elements are not allocated to registers, every access in the code to a matrix element becomes a memory access, so the total number of memory accesses is $4N^3$, where N is the dimension of the matrices. To simplify the computations, we will assume that the cache is fully associative, so there are no conflict misses. Thus, we need only consider cold and capacity misses. We see that there are no write misses in this code since every write to an element of

[4]http://icl.cs.utk.edu/papi/.

[5]The numbers reported by PAPI are the L2 cache accesses and misses from both application data accesses and from servicing of Translation Lookaside Buffer (TLB) misses. Some experimentation revealed that the TLB implementation on the Itanium 2 architecture triggers an L2 access for every TLB miss. Therefore, we measured the number of TLB misses and subtracted that number from the number of L2 cache accesses reported by PAPI to obtain an estimate for the number of L2 cache accesses from application data accesses. At larger problem sizes, we believe that TLB misses generate not only L2 accesses but L2 *misses* as well. We have no way of estimating this number, so at larger sizes, the miss rate is higher than we would expect if we were looking only at misses resulting from data accesses.

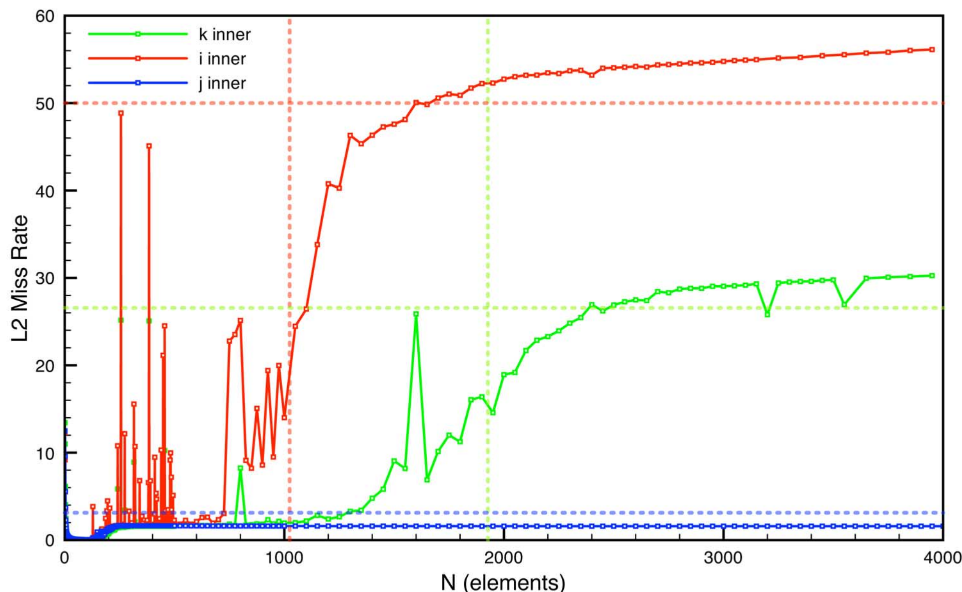

Fig. 2. ***MMM L2 cache miss ratio on Itanium 2.***

matrix C is preceded almost immediately by a read of the same element. Therefore, we can focus on the reads.

For small enough matrices, there are no capacity misses, so the only misses are cold misses. If we assume that the line size is b doubles, each miss brings in b elements of the matrix, so the number of cold misses is $3N^2/b$. Therefore, the miss ratio is $3N^2/4bN^3 = 3/4bN$. As N increases, the predicted miss ratio decreases until we start to observe capacity misses, which can be seen in Fig. 3.

Once N becomes large enough, there are capacity misses. For matrix multiply, there is a symmetry in the accesses made to the different elements of a given matrix, so the behavior is easy to analyze. When the problem size is small, the only misses are cold misses, and the accesses to that matrix enjoy both spatial and temporal locality, so the number of misses is N^2/b. As the problem size increases, the accesses to that matrix may lose temporal locality but may still enjoy spatial locality. In that case,

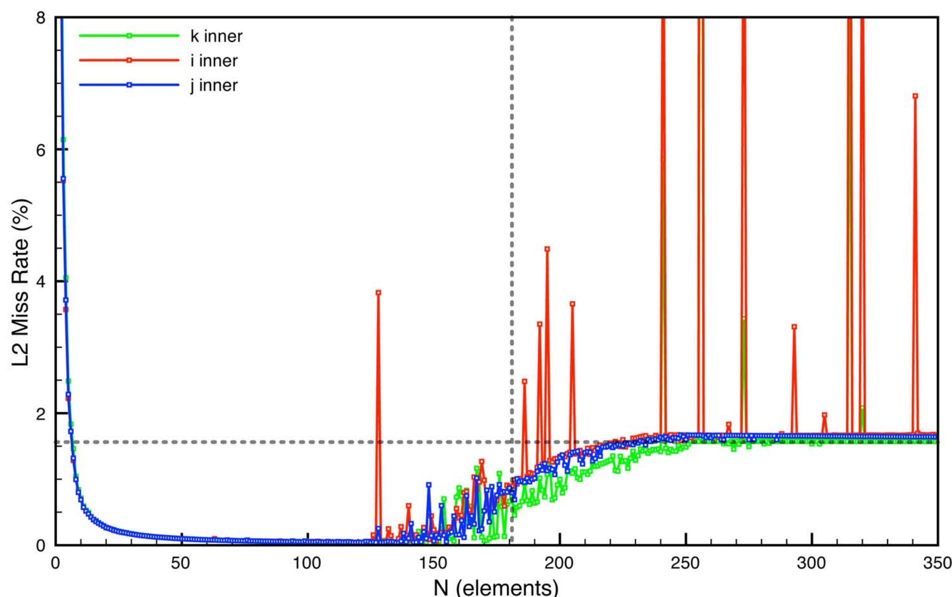

Fig. 3. ***MMM L2 cache miss ratio on Itanium 2 for small problem sizes.***

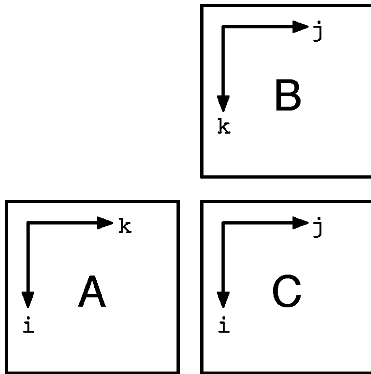

Fig. 4. ***Traversal order of the three matrices in MMM.***

one in every b accesses is a miss, so the number of misses is N^3/b. Finally, when the problem size increases even further, the accesses may have neither spatial nor temporal locality, and every access is a miss, so the number of misses is N^3. Therefore, the number of misses for a given matrix will be one of the three possibilities shown in Table 1.

We now want to estimate the problem sizes at which the matrices transition between these behaviors. This depends on the loop order and the layout of matrices in memory. We will focus on the *ijk* loop order and row-major data layout to be concrete. First, we introduce some notation that uses the well-known idea of *stack distance* for caches [17]: given two memory accesses in the address stream of a computation, the stack distance is the number of distinct cache lines touched by the computation between these accesses. Because of the symmetries in the accesses made by matrix multiplication, we can define the following concepts.

$d_t(\mathsf{M})$ the stack distance between successive accesses to a given element of matrix M.

$d_s(\mathsf{M})$ the minimum stack distance between successive accesses to distinct elements of matrix M that lie on the same cache line.

Consider a cache that can hold C doubles. Clearly, if C is less than $b * d_t(\mathsf{M})$, then accesses to M will not enjoy temporal locality. Between two successive accesses to a given element, more lines will be touched than can fit into cache, and hence the second access will miss.[6] Similarly, if $C < b * d_s(\mathsf{M})$, then M will not exhibit spatial locality.

We now compute d_t and d_s for each matrix for the *ijk* loop order.

A In the inner loop, A is walked in row-major order. Thus, $d_s(\mathsf{A})$ is three; accesses to the three matrices touch a cache line each. Next, we see that every j walks the same row. Thus, we return to the same memory location every N iterations. As a result, $d_t(\mathsf{A})$ is $N/b + N + 1$: N/b lines from A, N lines of B, as it is walked in column major order, and the single cache line of C that remains fixed. We can disregard terms in $d_t(\mathsf{A})$ and $d_s(\mathsf{A})$ that are not dependent on N, as they become negligible as problem size increases. Thus, A exhibits temporal locality for small problem sizes and has spatial locality at all sizes. We can therefore express the total misses for A in the *ijk* loop order as a function of problem size, line size, and cache size

$$\text{miss}_{ijk,A}(N,b,C) = \begin{cases} N^2/b & bN + N \le C \\ N^3/b & \text{otherwise.} \end{cases}$$

B We can easily see that an element of B is only touched once for each iteration of the i loop. Thus, we return to the same location once every N^2 iterations. In that time, we have touched all of B, a row of A, and a row of C. Therefore, $d_t(\mathsf{B}) = (N^2 + 2N)/b$. Because B is walked in column major order, we see that we do not return to the same cache line until we have walked all the way down the column (i.e., every N accesses). Thus, by a similar argument as for A, $d_s(\mathsf{B}) = N/b + N + 1$, giving us

$$\text{miss}_{ijk,B}(N,b,C) = \begin{cases} N^2/b & (N^2 + 2N) \le C \\ N^3/b & bN + N \le C \\ N^3 & \text{otherwise.} \end{cases}$$

C Here, we see that in the inner loop, the element of C we touch remains fixed. Thus, $d_t(\mathsf{C}) = 3$, as only the extra accesses to A and B intervene. We also see that the middle loop walks C in row-major order. Thus, $d_s(\mathsf{C}) = 3$ as well. This means that C always exhibits spatial and temporal locality, giving us

$$\text{miss}_{ijk,C}(N,b,C) = N^2/b.$$

[6] Due to Least Recently Used (LRU) replacement, $C \ge b * d_t(\mathsf{M})$ does not guarantee that M will exhibit temporal locality. However, the additional capacity required to ensure temporal locality is of lower order than d_t, so we ignore it in this analysis.

TABLE 1 Possible Miss Totals for a Given Matrix

Locality exhibited	Total misses
none	N^3
spatial	N^3/b
both	N^2/b

We can combine all of these miss totals, giving us a formula for the total number of misses for the *ijk* loop order

$$\text{miss}_{ijk}(N,b,C) = \begin{cases} 3N^2/b & (N^2+2N) \le C \\ N^3/b + N^2/b & bN + N \le C \\ (b+1)N^3/b & \text{otherwise.} \end{cases}$$

Some simple approximations allow us to model the miss ratio, given that the total number of accesses in MMM is $4N^3$:

$$\text{ratio}_{ijk}(N,b,C) = \begin{cases} 3/(4bN) & N \le \sqrt{(C)} \\ 1/(4b) & N \le C/(b+1) \\ (b+1)/(4b) & \text{otherwise.} \end{cases}$$

We have thus produced an analytical model that allows us to estimate the miss ratio for the *ijk* order of MMM for any cache parameters and problem size. For the Itanium 2 architecture ($C = 32\,K$ doubles, $b = 16$ doubles), this means that while $N \le 181$, we expect the miss ratio to decrease as N increases. Then, while $N \le 1927$, we expect a constant miss ratio of 1.6%. After that, we expect the miss ratio to jump to 25.6%. This matches closely with the experimental results shown in Figs. 2 and 3. The transitions between miss ratios are not immediate (unlike in our model) because the Itanium 2 does not have a fully associative cache. The asymptotic miss rate is higher than expected due to additional L2 misses caused by TLB accesses, as explained before.

Note that our model agrees with the simple approach presented to determining the miss rate in the absence of capacity misses [both find a miss rate of $3/(4bN)$]. However, our model provides a much more accurate estimate for when capacity misses begin to occur than a simpler model would. A naïve estimate is obtained by requiring that all three matrices fit in the cache, which can hold C doubles, which leads to the inequality $3N^2 \le C$, which can be simplified to

$$N \le \sqrt{(C/3)}.$$

This is a sufficient but not necessary condition since it is not necessary for all three matrices to reside in cache for the entire duration of the computation, as our model elucidates.

We can use this approach to model the miss ratio for other loop orders as well. Because the inner loop dominates the computation, we only present data and estimated miss ratios for two other orders *jki* and *kij*; the other three loop orders display similar behavior to the three presented orders, correspondent with their inner loop. We omit the derivations, which proceed along lines similar to those performed for the *ijk* order, and present only the miss ratio models

$$\text{ratio}_{jki}(N,b,C) = \begin{cases} 3/(4bN) & N \le \sqrt{(C)} \\ 1/(4b) & N \le C/(2b) \\ 1/2 & \text{otherwise} \end{cases}$$

$$\text{ratio}_{kij}(N,b,C) = \begin{cases} 3/(4bN) & N \le \sqrt{(C)} \\ 1/(4b) & N \le C/2 \\ 1/(2b) & \text{otherwise.} \end{cases}$$

There are several points of interest with these models.

- All the loop orders have the same miss ratio and "crossover point" when N is small; all three matrices exhibit both spatial and temporal locality. We call this the *small problem size* region of N, where no capacity misses are occurring. The estimated crossover point is represented as a vertical gray line in Fig. 3, and it corresponds with our experimental results.
- Each loop order has a *medium problem size* range of N, where some capacity misses are occurring, but not the maximum possible (in other words, one of the matrices is causing capacity misses but two other matrices still exhibit both spatial and temporal locality). The miss ratio in the medium data range is the same for all three loop orders. The estimated miss ratio is represented as a horizontal gray line in Fig. 3 and matches our experimental results.
- At some point for each loop order, two matrices start incurring capacity misses. Once this happens, the miss rate stays constant. We call this the *large problem size* region. The crossover point for each loop order is represented by a vertical line of the appropriate color[7] in Fig. 2. Once again, our predictions track closely with the experimental results.

  The expected final miss ratio for each order is represented by the horizontal line of the appropriate color. Here, our actual miss ratios are slightly higher (and continue to increase). We speculate that this is due to TLB effects, as discussed earlier.

B. Loop Transformations

Fig. 2 shows that if the matrices are stored in row-major order, the best loop order is one in which the *j* loop is innermost since this loop order exploits spatial locality

[7]There is no line for the *kij* order because the crossover point is at $N = 16\,K$, at which point running unoptimized MMM code becomes infeasible.

Original loop:
```
for i ∈ [0 : 1 : N − 1]
  ...
```
Strip mined loop:
```
for ii ∈ [0 : B : N − B]
  for i ∈ [ii : 1 : ii + B − 1]
    ...
```

Fig. 5. ***Loop strip mining.***

for two out of three matrices (B and C). If the loops in the input program are in some other order, a restructuring compiler can optimize performance by reordering loops appropriately. This transformation is called *loop permutation* [3] and is performed by most modern compilers.

To optimize performance further, we can exploit the fact that the miss ratio is minimized if the problem size is at the transition point between the small and medium problem size regions (that is, if the problem size is as large as possible without incurring capacity misses). If the full matrix multiplication can be decomposed into a sequence of smaller matrix multiplications of this size, the entire computation can be performed with this minimal miss ratio even if there is no reuse of data between small matrix multiplications.

The transformation that achieves this effect is called *loop tiling* or blocking, which is a combination of loop permutation and *strip mining*. Strip mining splits a single loop into a nested loop. The outer loop increments in steps of B and the inner loop performs B iterations. A simple example is in Fig. 5.

Strip mining by itself does not change the memory hierarchy behavior of the loop nest but can be combined with loop permutation to produce tiled code. For MMM, we can strip mine all three loops and then permute them as shown in Fig. 6.

After this transformation, the inner three loops now perform a standard MMM computation but over three much smaller matrices, each of size B^2. Thus, if B is chosen to minimize miss rate (i.e., $B = \sqrt{(C)}$), the overall computation will exhibit the minimum miss rate regardless of problem size. This tiling transformation can be done to optimize for multiple levels of the memory hierarchy. Loop tiling is supported in several modern compilers, including gcc4.

Original loop:
```
for i ∈ [0 : 1 : N − 1]
  for j ∈ [0 : 1 : N − 1]
    for k ∈ [0 : 1 : N − 1]
      C_ij = C_ij + A_ik * B_kj
```
Tiled loop:
```
for ii ∈ [0 : B : N − B]
  for jj ∈ [0 : B : N − B]
    for kk ∈ [0 : B : N − B]
      for i ∈ [ii : 1 : ii + B − 1]
        for j ∈ [jj : 1 : jj + B − 1]
          for k ∈ [kk : 1 : kk + B − 1]
            C_ij = C_ij + A_ik * B_kj
```

Fig. 6. ***Loop tiling.***

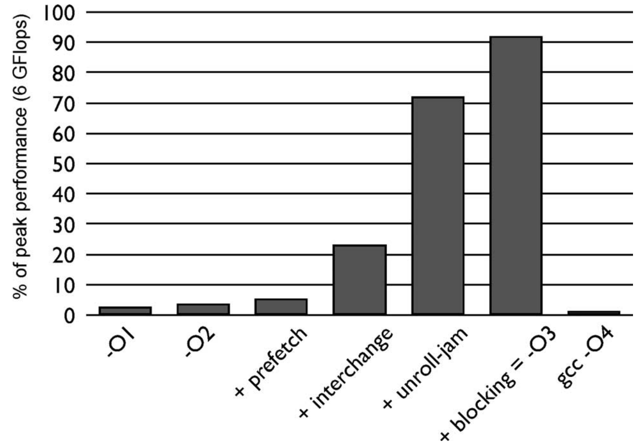

Fig. 7. ***Performance of Intel compiler on Itanium 2 when applying different transformations. With −O3, the compiler achieves 92% of peak performance (figure courtesy of W. Li, Intel Corp.).***

The Intel compiler for the Itanium 2 uses this approach to optimize code for the memory hierarchy of the Itanium [18]. Fig. 7 shows the results of these transformations. When all transformations are applied, the compiler is able to achieve 92% of peak performance,[8] demonstrating the efficacy of analytical models in producing high-performance code for dense linear algebra codes like MMM.

III. GENERATE-AND-TEST

Although analytical models can be effective, it is difficult to develop models that capture all aspects of machine architecture. For example, the models in Section II assumed that the cache is fully associative, but in reality, few caches are fully associative, so a complete accounting of misses must take conflict misses into account. On the other hand, once developed, analytical models can be used very efficiently in program optimization.

In some circumstances, it may be necessary to produce the best code possible even if it takes a very long time to do so. For example, we may be interested in generating a library that will be used repeatedly by many users, so it is reasonable to invest a lot of time in producing an optimized library. One solution is to use *library generators* based on the *generate-and-test* paradigm. To produce an optimized matrix–matrix multiplication (MMM) routine,

[8]These numbers are gathered using more refined variants of the techniques presented in [18].

for example, such a system would generate a large number of programs for performing MMM, run all of them on the actual machine, and select the one that gives the best performance. The generated programs usually implement the same algorithm but differ in the values they use for parameters such as tile sizes or loop unroll factors. In effect, the analytical techniques used in current compilers to derive optimal values for such parameters are replaced by a search over a suitably restricted space of parameter values. This self-tuning or *empirical optimization* approach is used by successful library generators such as ATLAS, which generates highly tuned BLAS, and FFTW [11] and SPIRAL [19], which generate FFT libraries.

A natural question is the following: how good is the code produced using analytical models compared to code produced by a generate-and-test system? This question is of more than academic interest because if analytical models are accurate enough, it may be possible to combine the use of models with local search techniques to generate very high-performance code without the cost of performing global searches over the space of parameter values.

Notice that it can be misleading to compare the performance of generated code generated by a compiler with the performance of code produced by the ATLAS system. Since the two systems use very different code generators, we cannot attribute differences in the performance of the generated codes to the use of models versus empirical search; it is possible that one system has a better code generator.

These issues are addressed more carefully in the study described below using the ATLAS library generator. The architecture of the ATLAS system is shown in Fig. 8. Like most systems that use empirical optimization, ATLAS has a module (called mmcase in this figure) that generates code, given certain parameter values. These parameters are described in more detail in Section III-A; for example, N_B is the tile size to be used when optimizing code for the L1 data cache. The search for parameters values is under the control of a module named mmsearch, described in more detail in Section III-B.

In general, there is an unbounded number of possible values for a parameter like N_B, so it is necessary to bound the size of the search space. When ATLAS is installed, it first runs a set of microbenchmarks to determine hardware parameters such as the capacity of the L1 data cache and the number of registers. These hardware parameters are used to bound the search space. The mmsearch module essentially enumerates points within this bounded search space, invokes the mmcase module to generate the appropriate code (denoted by mini-MMM in the figure), runs this code on the actual machine, and records its execution time. At the end of the search, the parameter values that gave the best performance are used to generate the library code. This library code is a simple subset of C, which can be viewed as portable assembly code, and it is compiled using a native compiler such as GCC to produce the final executable.

ATLAS can be modified by replacing the search module with a module (mmmodel) that uses analytical models of the type described in Section II to estimate optimal values for the optimization parameters. The precise model we use in this paper is described in Section III-C. This new architecture is shown at the bottom of Fig. 8. In general, model-driven optimization requires more knowledge of the architecture than empirical optimization; for example, the model-driven version of ATLAS in Fig. 8 needs to know the capacity of the L1 instruction cache to determine loop unrolling parameters. Since both ATLAS (referred to as ATLAS CGw/s) and the modified ATLAS (referred to as

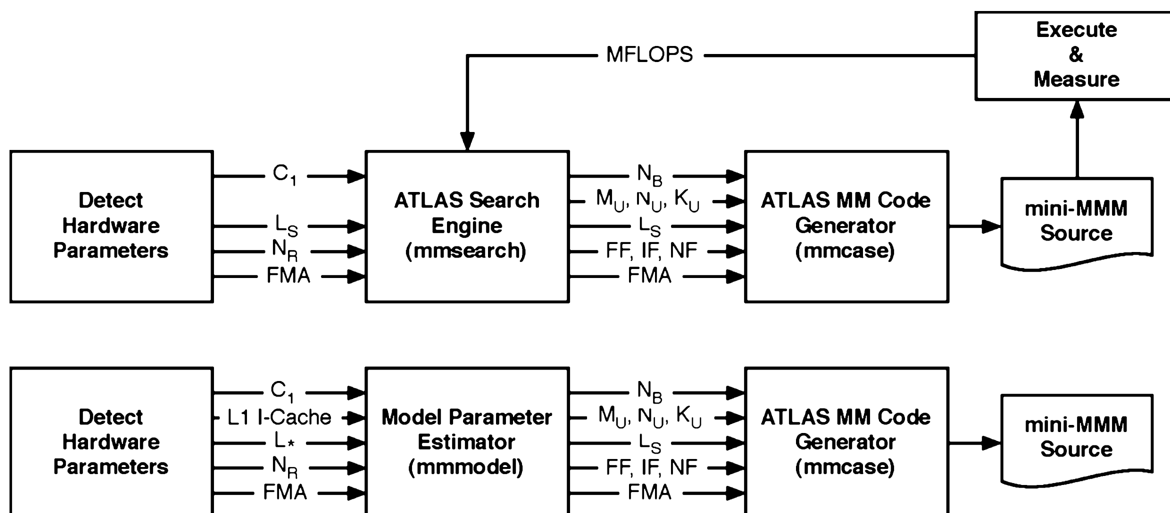

Fig. 8. ***Empirical optimization architecture.***

ATLAS Model) use the same code generator, we are assured that any difference in the performance of the generated code results solely from different choices for optimization parameter values.

A. ATLAS Code Generation

In this section, we use the framework of restructuring compilers to describe the structure of the code generated by the ATLAS Code Generator. While reading this description, it is important to keep in mind that ATLAS is not a compiler. Nevertheless, thinking in these terms helps clarify the significance of the code optimization parameters used in ATLAS. We focus on parameters relevant for memory hierarchy optimization.

Our discussion will use MMM, which is the key routine in the BLAS, as the running example. Naïve MMM code was shown in Fig. 1.

1) Memory Hierarchy Optimizations: This code can be optimized by tiling for the L1 data cache and registers, using approaches similar to those discussed in Section II-B.

- *Optimization for the L1 data cache:* To improve locality, ATLAS implements an MMM as a sequence of *mini-MMMs*, where each mini-MMM multiplies submatrices 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 loop transformations, as discussed in Section II-B, the triply nested loop of Fig. 1 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 submatrices of size $N_B \times N_B$, and we call this computation a *mini-MMM*.
- *Optimization for the register file:* ATLAS implements each mini-MMM as a sequence of *micro-MMMs*, where each micro-MMM multiplies an $M_U \times 1$ submatrix of A with a $1 \times N_U$ submatrix of B and accumulates the result into an $M_U \times N_U$ submatrix 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 $M_U + N_U + M_U \times N_U \leq (N_R - L_s)$, where N_R is the number of floating-point registers. L_s is a latency value that ATLAS uses when doing computation scheduling, which has the effect of reducing the number of available registers.
  In terms of loop transformations, 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 Fig. 9. 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, K, etc.). In reality, code should also be generated to handle the fractional tiles at the boundaries of the three arrays; we omit this *cleanup* code to avoid complicating the description. The k'' loop is unrolled completely. Fig. 10

```
// 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''
```

Fig. 9. ***MMM tiled for L1 data cache and registers.***

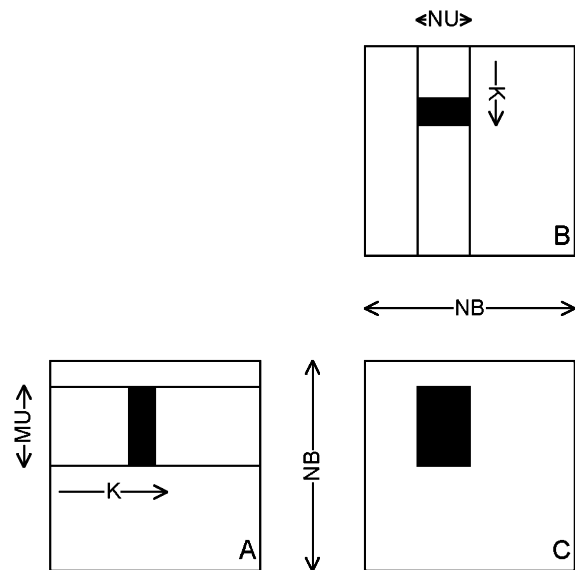

Fig. 10. ***Mini-MMM and micro-MMM.***

TABLE 2 Summary of Optimization Parameters

Name	Description
N_B	L1 data cache tile size
M_U	Register tile size
N_U	Register tile size
K_U	Unroll factor
L_s	Latency for computation scheduling

is a pictorial view of a mini-MMM computation within which a micro-MMM is shown using shaded rectangles.

2) Discussion: Table 2 lists the optimization parameters relevant to optimizing for the memory hierarchy.[9]

It is intuitively obvious that the performance of the generated mini-MMM code suffers if the values of the optimization parameters in Table 2 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. The goal therefore is to determine optimal values of these parameters for obtaining the best mini-MMM code.

B. Empirical Optimization in ATLAS CGw/s

ATLAS CGw/s performs a global search to determine optimal values for the optimization parameters listed in Table 2. In principle, the search space is unbounded because most of the parameters, such as N_B, are integers. Therefore, it is necessary to bound the search space using parameters of the machine hardware; for example, M_U and N_U, the dimensions of the register tile, must be less than number of registers.

Since ATLAS is self-tuning, it does not require the user to provide the values of such machine parameters; instead, it runs simple microbenchmarks to determine approximate values for these parameters. It then performs a global search, using the machine parameter values to bound the search space.

1) Measuring Machine Parameters: The machine parameters measured by ATLAS include, among others:

- C_1: the size of L1 data cache;
- N_R: the number of floating-point registers;
- L_s: the latency of floating-point multiplication;
- FMA: is there is a fused multiply–add operation?

The microbenchmarks used to measure these parameters are independent of matrix multiplication. For example, the microbenchmark for estimating C_1 is similar to the one discussed in Hennessy and Patterson [20].

[9] There are several other parameters used by the ATLAS Code generator (FF, IF, and NF in Fig. 8) that impact software pipelining and instruction scheduling. These are discussed in detail in [15].

2) Global Search for Optimization Parameter Values: To find optimal values for the optimization parameters in Table 2, ATLAS uses *orthogonal line search*, which finds an approximation to the optimal value of a function $y = f(x_1, x_2, \ldots, x_n)$, an n-dimensional optimization problem, by solving a sequence of n one-dimensional optimization problems corresponding to each of the n parameters. When optimizing the value of parameter x_i, it uses reference values for parameters $x_{i+1}, x_{i+2}, \ldots, x_n$, which have not yet been optimized. Orthogonal range search is an approximate method because it does not necessarily find the optimal value even for a convex function, but, with luck, 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 set of possible values considered during the optimization of each parameter, and iii) the reference value used for parameter k during the optimization of parameters $1, 2, \ldots, k-1$.

While ATLAS performs a search for numerous optimization parameters, we focus on those involved with the memory hierarchy: N_B, M_U and N_U. The optimization sequence used in ATLAS is the following.

1) *Find best N_B*
   In this step, ATLAS generates a number of mini-MMMs for matrix sizes $N_B \times N_B$, where N_B is a multiple of four that satisfies the following inequality:

$$16 \leq N_B \leq \min(80, \sqrt{C_1}). \quad (1)$$

   The reference values of M_U and N_U are set to the values closest to each other that satisfy

$$M_U \times N_U + M_U + N_U \leq (N_R - L_s). \quad (2)$$

   For each matrix size, ATLAS tries two extreme cases for K_U—no unrolling ($K_U = 1$) and full unrolling ($K_U = N_B$).
   The N_B value 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.
2) *Find best M_U N_U, K_U*
   This step is a straightforward search that refines the reference values of M_U and N_U that were used to find the best N_B. ATLAS tries all possible combinations of M_U and N_U that satisfy (2). Cases when M_U or N_U is one are treated specially. A test is performed to see if 1×9 unrolling or 9×1 unrolling is better than 3×3 unrolling. If not, unrolling factors of the form $1 \times U$ and $U \times 1$ for values of U greater than three are not checked.

C. Model-Based Optimization

Rather than using empirical search to find the optimal values for the parameters in Table 2, we can instead use analytical models such as those presented in Section II-A to estimate these values.

1) Measuring Hardware Parameters: Recall that the analytical models presented earlier are dependent on hardware parameters such as cache size and number of registers. Empirical optimizers use the values of machine parameters only to bound the search space, so approximate values for these parameters are adequate. In contrast, analytical models require accurate values for these parameters. Therefore, we have developed a tool called X-Ray [21] that accurately measures these values. In addition to the hardware parameters used by ATLAS, our model requires the capacity of the L1 instruction-cache to determine loop unrolling factors.

2) Estimating N_B: In our discussion of the ATLAS code generation module, we drew a connection between the optimization parameter N_B and the standard loop tiling transform. Knowing that N_B is the tile size for the outer tiling transform, we see that we would like to choose as large a tile size as possible without causing undue L1 cache misses.

This problem of finding the optimal tile size for the L1 cache is equivalent to finding the largest MMM problem size that is still in the *small problem size* region, as defined in Section II-A. Examining the models developed there, we find that this occurs when

$$N_B \approx \sqrt{(}C_1). \tag{3}$$

3) Estimating M_U and N_U: One can look at the register file as a software-controlled (optimal replacement), fully associative cache with unit line size and capacity equal to the number of available registers (N_R).

The innermost loops (the micro-MMM loop nest) perform a block computation, tuned for the register file rather than a cache. The ATLAS Code Generator uses the KJI loop order to tile for the register file, and thus we need to cache the complete $M_U \times N_U$ tile of C, an $1 \times N_U$ row of B and a single element of A. Therefore the analog of (3) for M_U and N_U is

$$M_U \times N_U + N_U + 1 \leq N_R. \tag{4}$$

Because the register file is software controlled, the ATLAS Code Generator is free to allocate registers differently than (4) prescribes. In fact, as discussed in Section III-A, it allocates a $M_U \times 1$ column of A, rather than a single element of A, to registers. Furthermore, it needs L_s registers to perform its software code scheduling, effectively reducing the number of available registers. Taking into account these details, we refine (4) to obtain (5)

$$M_U \times N_U + N_U + M_U \leq (N_R - L_s). \tag{5}$$

Because there are multiple possible values for M_U and N_U which satisfy (5), we begin by assuming that $N_U = M_U$, and maximize N_U. Having fixed N_U, we maximize M_U. Yotov *et al.* show how to estimate values for the other parameters needed by ATLAS [22]. For example, a value for K_U can be derived from the size of the L1 I-cache and the size of the compiled microkernel.

D. Experimental Results

We present here the results of running ATLAS CGw/s and ATLAS Model on ten common platforms. We also present numbers, where appropriate, for "ATLAS Unleashed." This variant of ATLAS makes use of hand-optimized MMM kernels. In addition to performing the standard empirical search, ATLAS Unleashed tests the performance of these hand-coded implementations and considers them as choices when finding the best performing implementation. While this does not provide any additional insight into the generate-and-test approach (since the "generation" was done by hand), we present these numbers to illustrate some of the remaining performance gap between self-tuning libraries and hand written code.

We did our experiments on the following platforms:

- RISC, out-of-order:
  - DEC Alpha 21264;
  - IBM Power3;
  - IBM Power4;
  - SGI R12K;
  - Sun UltraSPARC IIIi.
- EPIC, in-order:
  - Intel Itanium2.
- CISC, out-of-order:
  - AMD Opteron240;
  - AMD AthlonMP;
  - Intel Pentium III;
  - Intel Pentium 4.

The performance of the code generated by ATLAS Model and ATLAS Unleashed, normalized to the performance of the code generated by ATLAS CGw/S, is shown in Fig. 11. We see that on several platforms, ATLAS Model approaches ATLAS CGw/s in performance, despite performing no search at all. On several of the other platforms (specifically, the CISC machines), the basic model does not perform particularly well. However, this is because the model does not take into account the ability of out-of-order machines to rename registers and thus does not create large enough register tiles. A refined model that takes this into account was developed, and its performance also approaches that of ATLAS CGw/s [22].

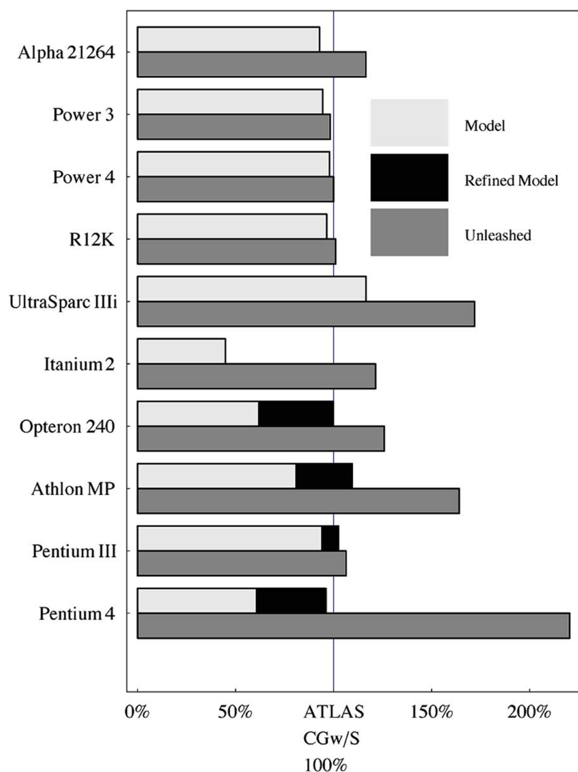

Fig. 11. *Summary of mini-MMM performance.*

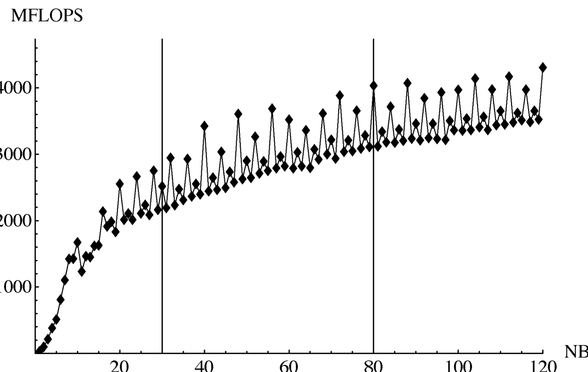

Fig. 12. *Sensitivity of Itanium 2 performance to N_B, near predicted optimal value.*

Itanium 2 Results: The analytical model performs worst for the Itanium 2. On this machine, the mini-MMM code produced by ATLAS Model is about 2.2 GFlops (55%) slower than mini-MMM code produced by ATLAS CGw/S. This is a rather substantial difference in performance, so it is necessary to perform sensitivity studies to understand the reasons why ATLAS Model is doing so poorly.

Fig. 12 shows that one reason for this difference is that ATLAS Model used $N_B = 30$, whereas ATLAS CGw/S used $N_B = 80$, the largest block size in the search space used by ATLAS CGw/S. When we studied these experimental results carefully, we realized that ATLAS Model was tiling for the L1 cache, since the X-Ray microbenchmarks determine the capacities of all cache levels, and the model used the capacity of the L1 cache to determine N_B. However, floating-point values are not cached in the L1 cache of the Itanium, as mentioned before, so the model should have determined the value of N_B for the L2 cache instead. Substituting the capacity of the L2 cache in the expression for N_B, we find $N_B = 180$, which gives slightly better performance than ATLAS CGw/S. ATLAS CGw/S does not find this point since it does not consider values of N_B above 80.

The Itanium results illustrate one advantage of empirical search over analytical models: the *lack* of intelligence of the search can occasionally provide better results when the platform behaves in a manner other than the model's expectation.

E. Discussion

There is a tradeoff between empirical search and analytical models: while empirical search offers the promise of eventually finding the best solution, we are often constrained by how long the search process takes and must stop before reaching optimality. The flip side is that analytical models can eliminate this search time entirely, choosing a particular solution immediately. However, imprecisions in the model (or faulty assumptions, as in the case of the Itanium 2) can lead to suboptimal parameter values' being chosen.

This suggests a hybrid approach: use the analytical model to "seed" the empirical search (for example, with parameters leading to tiling for the different levels of the memory hierarchy). A search space thus pruned can be covered with far fewer test implementations, and the search process will thus settle on the optimal parameters much faster. This approach is discussed in further detail in [23].

F. Summary

The generate-and-test method has the advantage that it does not require analytical models, so, in principle, it

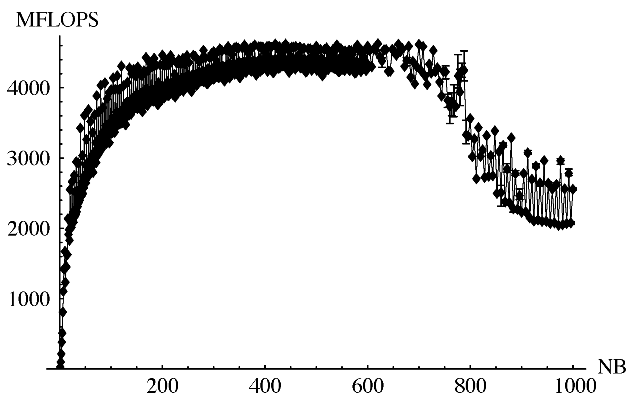

Fig. 13. *Sensitivity of Itanium 2 performance to N_B.*

combines performance with portability. As an alternative to generate-and-test, analytical models, augmented with local search if necessary, can be used to produce code with comparable performance while avoiding the expensive search process, at least for the BLAS. However, as we see with the Itanium 2 results, the assumptions made by an analytical model may be unsuitable for some architectures, in which case it is necessary to refine the model manually.

IV. CACHE-OBLIVIOUS APPROACH

The theoretical computer science community has investigated a different self-optimization strategy for memory hierarchies, based on the use of recursive, divide-and-conquer algorithms. To multiply two matrices A and B, we can divide one of the matrices (say, A) into two submatrices A_1 and A_2 and multiply A_1 and A_2 by B; the base case of this recursion is reached when both A and B have a single element. Programs written in this divide-and-conquer style perform *approximate* blocking in the following sense. Each division step generates subproblems of smaller size, and when the working set of some subproblem fits in a given level of cache, the computation can take place without suffering capacity misses at that level. The resulting blocking is only approximate since the size of the working set may be smaller than the capacity of the cache.

Although the initial work on this approach goes back to the 1981 paper of Hong and Kung [12], we are not aware of any studies that have compared the performance of highly tuned cache-conscious programs with that cache-oblivious programs. Is there a price for cache-obliviousness, and, if so, how much is it? In this section, we address this question, using matrix multiplication as the running example. First, we discuss why approximate blocking as performed by divide-and-conquer algorithms might work well by considering the effect of blocking on the latency of memory accesses and on the bandwidth required from memory. Then we give a detailed performance study of cache-oblivious programs for matrix multiplication on the Itanium 2.

A. Approximate Blocking

To examine the impact of blocking on the overhead from memory latency and bandwidth, we first consider a simple, two-level memory model consisting of one cache level and memory. The cache is of capacity C, with line size L_C, and has access latency l_C. The access latency of main memory is l_M. We consider blocked MMM, in which each block computation multiplies matrices of size $N_B \times N_B$. We assume conservatively that there is no data reuse between block computations.

We derive an upper bound on N_B by requiring the size of the working set of the block computation to be less than the capacity of the cache C. As discussed in Section II, the working set depends on the schedule of operations but is bounded above by the size of the subproblem. Therefore, the following inequality is a conservative approximation:

$$3N_B^2 \leq C. \tag{6}$$

a) Effect of blocking on latency: The total number of memory accesses each block computation makes is $4N_B^3$. Each block computation brings $3N_B^2$ data into the cache, which results in $3N_B^2/L_C$ cold misses. If the block size is chosen so that the working set fits in the cache and there are no conflict misses, the cache miss ratio of the complete block computation is $3/4N_B \times L_C$. Assuming that memory accesses are not overlapped, the expected memory access latency is as follows:

$$l = \left(1 - \frac{3}{4N_B \times L_C}\right) \times l_C + \frac{3}{4N_B \times L_C} \times l_M. \tag{7}$$

Equation (7) shows that the expected latency decreases with increasing N_B, so latency is minimized by choosing the largest N_B for which the working set fits in the cache. In practice, the expected memory latency computed from (7) is somewhat pessimistic because loads can be overlapped with each other or with actual computations, reducing the effective values of l_C and l_M.

b) Effect of blocking on bandwidth: Blocking also reduces the bandwidth required from memory. Each FMA operation in MMM reads three values and writes one value. The required bandwidth to perform these reads and writes is $4N^3 \div (N^3/2) = 8$ doubles/cycle. Fig. 14 shows the bandwidth between different levels of the memory hierarchy of the

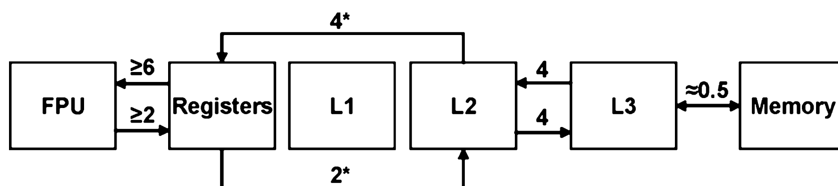

Fig. 14. ***Bandwidth of the Itanium 2 memory hierarchy, measured in doubles/cycle.** *Note: 1) Floating-point values are not cached at L1 in Itanium 2; they are transferred directly to/from L2 cache. 2) L2 cache can transfer four values to floating-point registers and two values from floating-point registers per cycle, but there is a maximum total of four memory operations.*

Itanium (floating-point values are not cached in the L1 cache on the Itanium). It can be seen that the register-file can sustain the required bandwidth but memory cannot.

To reduce the bandwidth required from memory, we can block the computation. Since each block computation requires $4N_B^2$ data moved, our simple memory model implies that the total data moved is $(N/N_B)^3 \times 4N_B^2 = 4N^3/N_B$. The ideal execution time of the computation is still $N^3/2$, so the bandwidth required from memory is $(4N^3/N_B) \div (N^3/2) = 8/N_B$ doubles/cycle. Therefore, cache blocking by a factor of N_B reduces the bandwidth required from memory by the same factor.

We can now write the following lower bound on the value of N_B, where $B(L1, M)$ is the bandwidth between cache and memory

$$\frac{8}{N_B} \leq B(L1, M). \tag{8}$$

Inequalities (6) and (8) imply the following inequality for N_B:

$$\frac{8}{B(L1, M)} \leq N_B \leq \sqrt{\frac{C}{3}}. \tag{9}$$

This argument generalizes to a multilevel memory hierarchy. If $B(L_i, L_{i+1})$ is the bandwidth between levels i and $i+1$ in the memory hierarchy, $N_B(i)$ is the block size for the ith cache level and C_i is the capacity of this cache, we obtain the following inequality:

$$\frac{8}{B(L_i, L_{i+1})} \leq N_B(i) \leq \sqrt{\frac{C_i}{3}}. \tag{10}$$

In principle, there may be no values of $N_B(i)$ that satisfy the inequality. This can happen if the capacity of the cache as well as the bandwidth to the next level of the memory hierarchy are small. According to this model, the bandwidth problem for such problems cannot be solved by blocking (in principle, there may be other bottlenecks such as the inability of the processor to sustain a sufficient number of outstanding memory requests). An early version of the Power PC 604 suffered from this problem—it had an L1 cache of only 4K double words, and there was not enough bandwidth between the L1 and L2 caches to keep the multiply–add unit running at peak.[10]

For the Itanium 2, we have seen that register blocking is needed to prevent the bandwidth between registers and L2 cache from becoming the bottleneck. If $N_B(R)$ is the size of the register block, we see that $8/4 \leq N_B(R) \leq \sqrt{126/3}$. Therefore, $N_B(R)$ values between two and six will suffice. If we use $N_B(R)$ in this range, the bandwidth required from L2 to registers is between 1.33 and four doubles per cycle. Note that this much bandwidth is also available between the L2 and L3 caches. Therefore, it is not necessary to block for the L2 cache to ameliorate bandwidth problems. Unfortunately, this bandwidth exceeds the bandwidth between L3 cache and memory. Therefore, we need to block for the L3 cache. The appropriate inequality is $8/0.5 \leq N_B(L_3) \leq \sqrt{4\text{ MB}/3}$. Therefore, $N_B(L_3)$ values between 16 and 418 will suffice.

Thus, for the Itanium 2, there is a range of block sizes that can be used. Since the upper bound in each range is more than twice the lower bound, the approximate blocking of a divide-and-conquer implementation of a cache-oblivious program will generate subproblems in these ranges, and therefore bandwidth is not a constraint. Of course, latency of memory accesses may still be a problem. In particular, since blocking by cache-oblivious programs is only approximate, (7) suggests that reducing the impact of memory latency is more critical for cache-oblivious codes than it is for cache-conscious codes.

Fig. 16 shows the results of performing MMMs of various sizes on the Itanium. Since we explore a large number of implementations in this section, we use a tuple to distinguish them, the first part of which describes the outer control structure:

- R—using the *recursive* control structure;
- I—using a triply nested *iterative* control structure;

The second part of the tuple describes the microkernel, which will be explained as the microkernels are developed in Section IV-B. However, when the outer control structure invokes a single statement to perform the computations, we use the symbol S (for *statement*). For completeness, we include performance lines for MMM performed by the vendor BLAS using standard row-major storage format for the arrays.

With this notation, note that the lines labeled RS in Fig. 16 show the performance of the cache-oblivious program, while the lines labeled IS show the performance of the nested loop program. Both programs perform very poorly, obtaining roughly 1% of peak on all the machines. As a point of comparison, vendor BLAS on the Itanium 2 achieves close to peak performance.

B. Microkernels

To improve the efficiency of the recursive program, it is necessary to terminate the recursion when the problem size becomes smaller than some value and invoke a long basic block of operations called a microkernel that is obtained by unrolling the recursive code completely for a problem of size $R_U \times R_U \times R_U$ [14]. The overall recursive program invokes this microkernel as its base case. There are two advantages to this approach. First, it is possible to

[10]We thank Fred Gustavson for bringing this example to our attention.

perform register allocation and scheduling of operations in the microkernel, thereby exploiting registers and the processor pipeline. Secondly, the overhead of recursion is reduced. We call the long basic block a *recursive* microkernel since the multiply–add operations are performed in the same order as they were in the original recursive code (there is no recursion in the code of the microkernel, of course). The optimal value of R_U is determined empirically for values between one and 15.

Together with the control structure, one needs to worry about which data structure to use to represent the matrices. Virtually all high-performance BLAS libraries internally use a form of a blocked matrix, such as row-block-row (RBR). An alternative is to use a recursive data layout, such as a space-filling curve like Morton-Z [24]. We compared the MMM performance using both these choices and rarely saw any performance improvement using Morton-Z order over RBR. Thus we use RBR in all experiments in this paper, and we chose the data block size to match our kernel block size. Note, however, that the native BLAS on all the machines use standard row-major order for the input arrays and copy these arrays internally into RBR format, so care should be taken in performance comparisons with the native BLAS.

Since the performance of the overall MMM depends critically on the performance of the microkernel, we implemented an integrated instruction scheduler and register allocator to generate code for the microkernel [16].

Fig. 15 shows the performance of recursive microkernels of different sizes. A number of different strategies were used to compile the microkernels. The line labeled BC shows the performance when the integrated instruction scheduler and register allocator are used. It can be seen that the performance of the microkernel in isolation (that is, when the problem size is small enough that there are no cache misses) is roughly 93% of peak. Although this level of performance is not sustained in the complete MMM, the line labeled RR (recursive outer control structure and recursive microkernel) in Fig. 16 shows that the complete MMM reaches about 3.8 Gflops or about 63% of peak. This performance is well below the performance of the vendor BLAS library. Experiments on other architectures show even larger gaps between the performance of the recursive code and the best library code; for example, on the UltraSPARC III, the recursive code gave only 38% of peak performance, while the library code ran at 85% of peak.

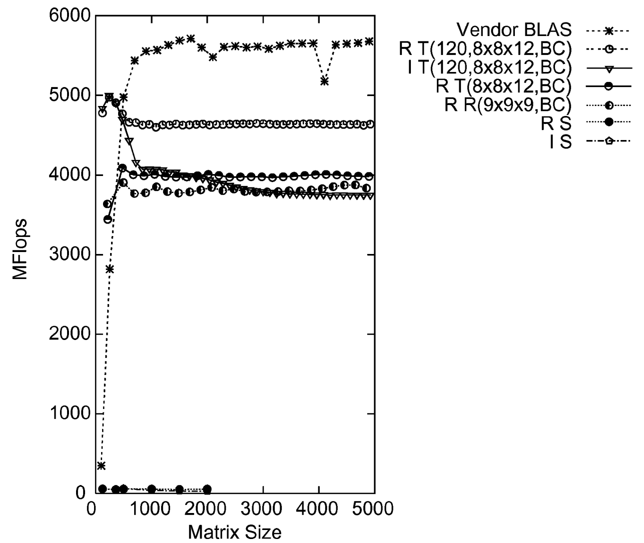

Fig. 16. ***Full MMM performance on Itanium 2.***

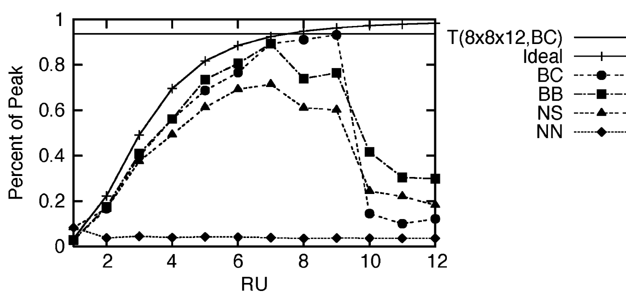

Fig. 15. ***Microkernel performance on Itanium 2.***

To get some insight into these performance differences, we replaced the recursive microkernel described in this section with an iterative microkernel, similar to the one used in ATLAS, that was described in Section III. The outer control structure is still recursive but invokes the iterative microkernel rather than the recursive microkernel when the problem size is small enough. In Fig. 15, the line labelled $\mathbf{T}(\mathbf{8}\times\mathbf{8}\times\mathbf{12})$ shows the performance of the iterative microkernel in isolation (the M_U, N_U, K_U parameters are 8, 8, and 12, respectively). It can be seen that the performance of both microkernels is roughly equal. Not surprisingly, replacing the recursive microkernel with the iterative microkernel does not change overall performance very much, as can be seen in the line labeled RT in Fig. 16. We believe that recursive microkernels perform well on the Itanium because the architecture has enough registers that the sizes of the recursive and iterative microkernels are comparable ($\mathbf{8}\times\mathbf{8}\times\mathbf{8}$ versus $\mathbf{8}\times\mathbf{8}\times\mathbf{12}$). On all other architectures, iterative microkernels were significantly larger than recursive microkernels; for example, on the UltraSPARC III, the sizes of the recursive and iterative microkernels were $\mathbf{4}\times\mathbf{4}\times\mathbf{4}$ and $\mathbf{4}\times\mathbf{4}\times\mathbf{120}$, respectively, leading to significant performance differences.

C. Explicit Cache Tiling

To determine the advantage of explicit cache tiling, we generated code that was tiled explicitly for the L2 cache on the Itanium and invoked the iterative microkernel discussed in Section IV-B. We used an L2 cache tile size

of $\mathbf{120} \times \mathbf{120} \times \mathbf{120}$, and we refer to the tile computation as an *iterative minikernel*. In Fig. 16, the performance of this version is labelled $\mathbf{RT}(\mathbf{120}, \mathbf{8} \times \mathbf{8} \times \mathbf{12})$. It can be seen that explicitly tiling for the L2 cache improves performance by roughly 1 GFlop. Intuitively, this version uses recursion to tile approximately for the L3 cache and explicit tiling to tile for the L2 cache.

To determine the importance of tiling for the L3 cache, we also wrote a simple triply nested loop that invoked the minikernel repeatedly. Intuitively, this version is tiled only for the L2 cache. The performance of this version is labeled $\mathbf{IT}(\mathbf{120}, \mathbf{8} \times \mathbf{8} \times \mathbf{12})$ in Fig. 16. Performance starts out at roughly 5 GFlops but drops once the working set overflows the L3 cache, demonstrating the importance of tiling for the L3 cache.

Since the vendor BLAS on most platforms is proprietary, it is difficult to determine the reason for the remaining gap in performance between vendor BLAS and the recursive version that invokes the minikernel $\mathbf{RT}(\mathbf{120}, \mathbf{8} \times \mathbf{8} \times \mathbf{12})$. Explicitly tiling for the L3 cache may reduce some of this performance gap. Many BLAS implementations also perform prefetching to reduce memory latency, and it is possible that this would eliminate the remaining performance gap.

D. Summary

We conclude that although self-optimization for memory hierarchies using the divide-and-conquer approach is an attractive idea, the performance of the resulting code may not be as good as that of carefully tuned code even on a machine like the Itanium that has a lot of registers. There appear to be a number of reasons for this. On most architectures, recursive microkernels perform significantly worse than iterative microkernels, so it is necessary to step outside the divide-and-conquer paradigm to produce microkernels that do a good job of exploiting registers and processor pipelines. Furthermore, explicit cache tiling seems to perform better than approximate cache tiling using divide-and-conquer. Finally, prefetching to reduce memory latency is well understood in the context of iterative codes but is not well understood for recursive codes.

V. CONCLUSIONS

In this paper, we presented a simple analytical model of the memory hierarchy behavior of matrix-multiplication and showed how it can be used to compile efficient code for a modern high-performance processor like the Itanium 2. We also discussed two self-optimization approaches that do not require analytical models: the generate-and-test approach and cache-oblivious approach. Using a modified version of the ATLAS system, we showed that although analytical models are necessarily simple, they can be competitive with the generate-and-test approach in ATLAS and therefore can be used to avoid a global search over the optimization parameter space. We also showed that the code produced by the cache-oblivious approach may not perform as well as highly tuned cache-conscious code and suggested a number of directions for improving the performance of cache-oblivious code. ■

Acknowledgment

The authors would like to acknowledge the contributions of a number of people who participated in the work discussed here and reported in earlier papers: M. Garzaran, J. Gunnels, F. Gustavson, X. Li, D. Padua, G. Ren, T. Roeder, P. Stodghill, and K. Yotov.

REFERENCES

[1] D. Padua and M. Wolfe, "Advanced compiler optimization for supercomputers," *Commun. ACM*, vol. 29, pp. 1184–1201, Dec. 1986.

[2] P. Feautrier, "Some efficient solutions to the affine scheduling problem—Part 1: One dimensional time," *Int. J. Parallel Program.*, Oct. 1992.

[3] R. Allan and K. Kennedy, *Optimizing Compilers for Modern Architectures*. San Mateo, CA: Morgan Kaufmann, 2002.

[4] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proc. 3rd SIAM Conf. Parallel Process. Sci. Comput.*, Dec. 1987.

[5] I. Kodukula, N. Ahmed, and K. Pingali, "Data-centric multi-level blocking," in *Proc. ACM Program. Lang., Design Implement. (SIGPLAN)*, Jun. 1997.

[6] I. Kodukula and K. Pingali, "Imperfectly nested loop transformations for memory hierarchy management," in *Proc. Int. Conf. Supercomput.*, Rhodes, Greece, Jun. 1999.

[7] U. Banerjee, "A theory of loop permutations," in *Lang. Compil. Parallel Comput.*, 1989, pp. 54–74.

[8] M. Wolf and M. Lam, "A data locality optimizing algorithm," in *Proc. Conf. Program. Lang. Design Implement. (SIGPLAN 1991)*, Jun. 1991.

[9] W. Li and K. Pingali, "Access normalization: Loop restructuring for NUMA compilers," *ACM Trans. Comput. Syst.*, 1993.

[10] M. Cierniak and W. Li, "Unifying data and control transformations for distributed shared memory machines," in *Proc. Conf. Program. Lang. Design Implement. (SIGPLAN 1995)*, Jun. 1995.

[11] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, Seattle, WA, May 1998, vol. 3, pp. 1381–1384.

[12] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Proc. 13th Annu. ACM Symp. Theory Comput.*, 1981, pp. 326–333.

[13] F. Gustavson, "Recursion leads to automatic variable blocking for dense linear-algebra algorithms," *IBM J. Res. Develop.*, vol. 41, no. 6, pp. 737–755, 1997.

[14] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. 40th Annu. Symp. Found. Comput. Sci. (FOCS '99)*, 1999, p. 285.

[15] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill, "Is search really necessary to generate high-performance BLAS?" *Proc. IEEE*, vol. 93, no. 2, 2005.

[16] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-aware programs," in *Proc. 19th Annu. ACM Symp. Parallel. Algorithms Architect. (SPAA)*, 2007.

[17] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–92, 1970.

[18] S. Ghosh, A. Kanhere, R. Krishnaiyer, D. Kulkarni, W. Li, C. C. Lim, and J. Ng, "Integrating high-level optimizations in a production compiler: Design and implementation experience," in *Proc. Compil. Construct.*, 2003.

[19] J. Johnson, R. W. Johnson, D. A. Padua, and J. Xiong, "Searching for the best FFT formulas with the SPL compiler," *Lecture Notes in Computer Science*, vol. 2017, p. 112, 2001.

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990.

[21] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," in *Proc. 2005 Int. Conf. Meas. Model. Comput. Syst. (SIGMETRICS'05)*, 2005.

[22] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu, "A comparison of empirical and model-driven optimization," in *Proc. ACM Conf. Program. Lang. Design Implement. (SIGPLAN 2003)*, 2003, pp. 63–76.

[23] A. Epshteyn, M. Garzaran, G. DeJong, D. A. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali, "Analytical models and empirical search: A hybrid approach to code optimization," in *Proc. 18th Int. Workshop Lang. Compil. Parallel Comput. (LCPC)*, 2005.

[24] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi, "Recursive array layouts and fast parallel matrix multiplication," in *Proc. ACM Symp. Parallel Algorithms Architect.*, 1999, pp. 222–231.

ABOUT THE AUTHORS

Milind Kulkarni graduated in 2002 with a B.S. in Computer Science and Computer Engineering from North Carolina State University. He then joined the Ph.D. program at Cornell University, where he has worked with Professor Keshav Pingali since 2003. His research focuses on language, compiler and runtime techniques for parallelizing irregular applications. He is currently based out of the University of Texas at Austin.

Keshav Pingali (Member, IEEE) is the W.A."Tex" Moncrief Chair of Computing in the Department of Computer Sciences at the University of Texas, Austin. He received the B.Tech. degree in Electrical Engineering from IIT, Kanpur, India in 1978, the S.M. and E.E. degrees from MIT in 1983, and the Sc.D. degree from MIT in 1986. He was on the faculty of the Department of Computer Science at Cornell University from 1986 to 2006, where he held the India Chair of Computer Science. Among other awards, he has won the I.I.T. Kanpur President's Gold Medal (1978), the Presidential Young Investigator's Award (1989), and the Russell Teaching Award of the College of Arts and Sciences at Cornell University (1998).