Is Search Really Necessary to Generate High-Performance BLAS?

Kamen Yotov, Xiaoming Li, Gang Ren, Maria Garzaran,
David Padua, Keshav Pingali, Paul Stodghill

***Abstract*— A key step in program optimization is the estimation of optimal values for parameters such as tile sizes and loop unrolling factors. Traditional compilers use simple analytical models to compute these values. In contrast, library generators like ATLAS use global search over the space of parameter values by generating programs with many different combinations of parameter values, and running them on the actual hardware to determine which values give the best performance. It is widely believed that traditional model-driven optimization cannot compete with search-based empirical optimization because tractable analytical models cannot capture all the complexities of modern high-performance architectures, but few quantitative comparisons have been done to date.**

To make such a comparison, we replaced the global search engine in ATLAS with a model-driven optimization engine, and measured the relative performance of the code produced by the two systems on a variety of architectures. Since both systems use the same code generator, any differences in the performance of the code produced by the two systems can come only from differences in optimization parameter values. Our experiments show that model-driven optimization can be surprisingly effective, and can generate code with performance comparable to that of code generated by ATLAS using global search.

***Index Terms*— program optimization, empirical optimization, model-driven optimization, compilers, library generators, BLAS, high-performance computing**

I. Introduction

The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.

John Von Neumann

It is a fact universally recognized that current restructuring compilers do not generate code that can compete with hand-tuned code in efficiency, even for a simple kernel like matrix multiplication. This inadequacy of current compilers does not stem from a lack of technology for transforming high-level programs into programs that run efficiently on modern high-performance architectures; over the years, the compiler community has invented innumerable techniques such as linear loop transformations [5, 11, 14, 29, 42], loop tiling [27, 28, 43] and loop unrolling [4, 32] for enhancing locality and parallelism. Other work has focused on algorithms for estimating optimal values for parameters associated with these transformations, such as tile sizes [7, 13, 36] and loop unroll factors [4]. Nevertheless, performance-conscious programmers must still optimize their programs manually [15, 19].

The simplest manual approach to tuning a program for a given platform is to write different versions of that program, evaluate the performance of these versions on the target platform, and select the one that gives the best performance. These different versions usually implement the same algorithm, but differ in the values they use for parameters such as tile sizes and loop unroll factors. The architectural insights and domain knowledge of the programmer are used to limit the number of versions that are evaluated. In effect, the analytical techniques used in current compilers to derive optimal values for such parameters are replaced by an *empirical search* over a suitably restricted space of parameter values (by empirical search, we mean a three step process: (1) generating a version of the program corresponding to each combination of the parameters under consideration, (2) executing each version on the target machine and measuring its performance, and (3) selecting the version that performs best). This approach has been advocated most forcefully by Fred Gustavson and his co-workers at IBM, who have used it for many years to generate the highly optimized ESSL and PESSL libraries for IBM machines [34]. Recently, a number of projects such as FFTW [17, 18], PhiPAC [2, 6], ATLAS [1, 41], and SPIRAL [26, 33] have automated the generation of the different program versions whose performance must be evaluated. Experience shows that these library generators produce much better code than native compilers do on modern high-performance architectures.

Our work was motivated by a desire to understand the performance gap between the BLAS codes produced by ATLAS and by restructuring compilers, with the ultimate goal of improving the state of the art of current compilers. One reason why compilers might be at a disadvantage is that they are general-purpose and must be able to optimize any program, whereas a library generator like ATLAS can focus on a particular problem domain. However, this is somewhat implausible because dense numerical linear algebra, the particular problem domain of ATLAS, is precisely the area that has been studied most intensely by the compiler community, and there is an extensive collection of well-understood transformations for optimizing dense linear algebra programs. Another reason for the inadequacy of current compilers might be that new transformations, unknown to the compiler community, are required

This work was supported by NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-012140. K. Yotov, K. Pingali and P. Stodghill are with Cornell University; X. Li, G. Ren, M. Garzaran and D. Padua are with University of Illinois at Urbana-Champaign

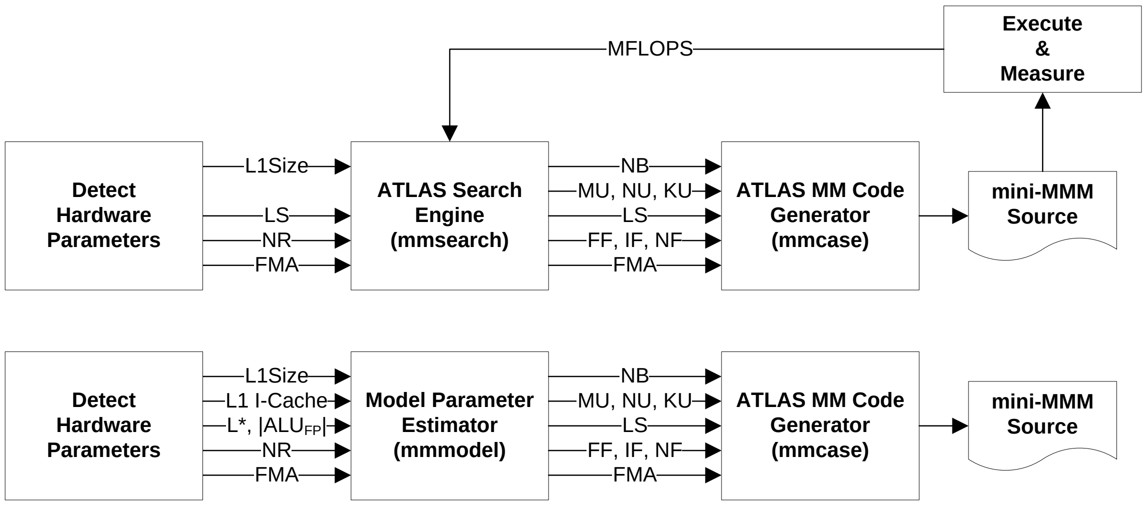
Execute & Measure
MFLOPS
Detect Hardware Parameters
L1Size
LS
NR
FMA
ATLAS Search Engine (mmsearch)
NB
MU, NU, KU
LS
FF, IF, NF
FMA
ATLAS MM Code Generator (mmcase)
mini-MMM Source
Detect Hardware Parameters
L1Size
L1 I-Cache
L*, |ALU_FP|
NR
FMA
Model Parameter Estimator (mmmodel)
NB
MU, NU, KU
LS
FF, IF, NF
FMA
ATLAS MM Code Generator (mmcase)
mini-MMM Source

Fig. 1. Architecture of ATLAS and of Model-driven ATLAS

to produce code of the same quality as the code produced by ATLAS. Finally, it is possible that the analytical models used by compilers to estimate optimal values for transformation parameters are overly simplistic, given the complex hardware of modern computers, so they are not able to produce good values for program optimization parameters.

No definitive studies exist to settle these matters. Our research is the first quantitative study of these issues.

Figure 1 shows our experimental set-up, which makes use of the original ATLAS system (top of the figure) and a modified version (bottom of the figure) that uses analytical models instead of empirical search. Like any system that uses empirical search, ATLAS has (i) a module that controls the search, which is used to determine optimal values for code optimization parameters (mmsearch), and (ii) a module that generates code, given these values (mmcase). The parameters used by ATLAS are described in more detail in Section II; for example, N_B is the tile size to be used when optimizing code for the L1 data cache. 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 micro-benchmarks 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 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 is coded in a simple subset of C, which can be viewed as portable assembly code, and it is compiled to produce the final executable.

We first studied the code generation module[1], and determined that the code it produces can be viewed as the end result of applying standard compiler transformations to high-level BLAS codes. As we describe in Section II, the code produced by ATLAS is similar to what we would get if we applied cache tiling, register tiling, and operation scheduling to the standard three-loop matrix multiplication code. This exercise ruled out the possibility that ATLAS incorporated some transformation, unknown to the compiler community, that was critical for obtaining good performance. We then modified ATLAS by replacing the search module, described in more detail in Section III, with a module (mmmodel) that uses standard analytical models to estimate optimal values for the optimization parameters, as described in Section IV. Since both ATLAS and the modified ATLAS 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. In Section V, we present experimental results on ten different platforms, comparing

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

Our results show that on all ten platforms, a relatively simple and very intuitive model is able to estimate near-optimal values for the optimization parameters used by the ATLAS Code Generator. We conclude in Section VI with a discussion of our main findings, and suggest future directions for research.

One feature of ATLAS is that it can make use of hand-tuned BLAS routines, many of which are included in the ATLAS distribution. When ATLAS is installed on a machine, these hand-coded routines are executed and evaluated. If the performance of one of these hand-coded routines surpasses the performance of the code generated by the ATLAS Code Generator, the hand-coded routine is used to produce the library. For example, neither the ATLAS Code Generator nor the C compilers on the Pentium IV exploit the SSE2 vector extensions to the x86 instruction set, so ATLAS-generated matrix multiplication code on the Pentium IV runs at around

[1]The description of ATLAS in this paper was arrived at by studying the ATLAS source code. In case of any discrepancy between this description and how the ATLAS system is actually implemented, the documentation of the ATLAS project should be considered to be authoritative [39–41].

1.5 GFLOPS. However, the matrix multiplication routine in the library produced by ATLAS runs at 3.3 GFLOPS because it uses carefully hand-coded kernels, contributed by expert programmers and part of the ATLAS distribution, which use these vector extensions.

Our concern in this paper is not with handwritten code, but with the code produced by the ATLAS Code Generator and with the estimation of optimal values for the parameters that are inputs to the code generator. To make clear distinctions, we use the following terminology in the rest of this paper.

- *ATLAS CGw/S:* This refers to the ATLAS system in which all code is produced by the *ATLAS Code Generator with Search* to determine parameter values. No handwritten, contributed code is allowed.
- *ATLAS Model:* This refers to the modified ATLAS system we built in which all code is produced by the ATLAS Code Generator, using parameter values produced from analytical models.
- *ATLAS Unleashed:* This refers to the complete ATLAS distribution which may use hand-written codes and predefined parameter values (*architectural defaults*) to produce the library. Where appropriate, we include, for completeness, the performance graphs for the libraries produced by ATLAS Unleashed.

II. ATLAS Code Generator

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 concentrate on matrix-matrix multiplication (MMM), which is the key routine in the BLAS. Naïve MMM code is shown in Figure 2. In this, and all later codes, we use the MATLAB notation $[First : Step : Last]$ to represent the set of all integers between $First$ and $Last$ in steps of $Step$.

```
for $i \in [0:1:N-1]$
    for $j \in [0:1:M-1]$
        for $k \in [0:1:K-1]$
            $C_{ij} = C_{ij} + A_{ik} \times B_{kj}$
```

Fig. 2. Naïve MMM Code

A. Memory Hierarchy Optimizations

The code shown in Figure 2 can be optimized for locality by blocking for the L1 data cache and registers. Blocking is an algorithmic transformation that converts the matrix multiplication into a sequence of small matrix multiplications, each of which multiplies small blocks of the original matrices. Blocking matrix multiplication for memory hierarchies was discussed by McKellar and Coffman as early as 1969 [31]. The effect of blocking can be accomplished by a loop transformation called tiling, which was introduced by Wolfe in 1987 [43].

- *Optimization for the L1 data cache*:
  ATLAS implements an MMM as a sequence of *mini-MMMs*, where each mini-MMM multiplies sub-matrices of size $N_B \times N_B$. N_B is an optimization parameter whose value must be chosen so that the working set of the mini-MMM fits in the cache.
  In the terminology of restructuring compilers, the triply-nested loop of Figure 2 is tiled with tiles of size $N_B \times N_B \times N_B$, producing an *outer* and an *inner* loop nest. For the outer loop nest, code for both the JIK and IJK loop orders are implemented. When the MMM library routine is called, it uses the shapes of the input arrays to decide which version to invoke, as described later in this section. For the inner loop nest, only the JIK loop order is used, with (j', i', k') as control variables. This inner loop nest multiplies sub-matrices of size $N_B \times N_B$, and we call this computation a *mini-MMM*.
- *Optimization for the register file*: ATLAS represents each mini-MMM into a sequence of *micro-MMMs*, where each micro-MMM multiplies an $M_U \times 1$ sub-matrix of A by 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 without floating-point register spills. For this to happen, it is necessary that $M_U + N_U + M_U \times N_U \leq N_R$, where N_R is the number of floating-point registers.
  In terms of restructuring compiler terminology, the (j', i', k') loops of the mini-MMM from the previous step are tiled with tiles of size $N_U \times M_U \times K_U$, producing an extra (*inner*) loop nest. The JIK loop order is chosen for the outer loop nest after tiling, and the KJI loop order for the loop nest of the mini-MMM after tiling.
  The resulting code after the two tiling steps is shown in Figure 3. To keep this code simple, we have assumed that all step sizes in these loops divide the appropriate loop bounds exactly (so N_B divides M, N, and K, etc.). In reality, code should also be generated to handle the fractional tiles at the boundaries of the three arrays; we omit this *clean-up* code to avoid complicating the description. The strategy used by ATLAS to copy blocks of the arrays into contiguous storage is discussed later in this section. Figure 4 is a pictorial view of a mini-MMM computation within which a micro-MMM is shown using shaded rectangles. In this figure, the values assigned to variable K are produced by executing the two `for` loops in Figure 3 corresponding to indices k' and k''.

To perform register allocation for the array variables referenced in the micro-MMM code, ATLAS uses techniques similar to those presented in [8]: the micro-MMM loop nest (j'', i'') in Figure 3 is fully unrolled, producing $M_U \times N_U$ multiply and add statements in the body of the middle loop nest. In the unrolled loop body, each array element is accessed several times. To enable register allocation of these array elements, ATLAS uses scalar replacement [9] to introduce a scalar temporary for each element of A, B, and C that is referenced in the unrolled micro-MMM code, and replaces array references in the unrolled micro-MMM code with references to these

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

Fig. 3. MMM tiled for L1 data cache and Registers

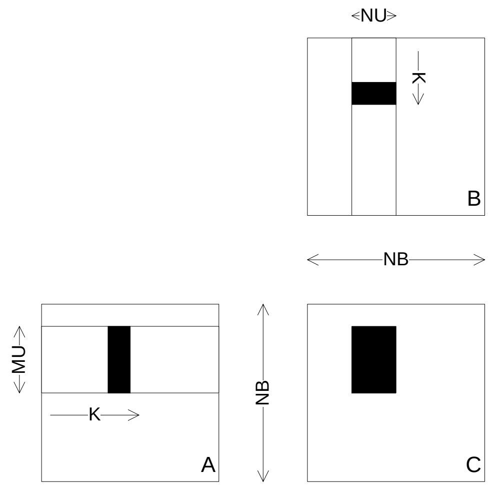

Fig. 4. mini-MMM and micro-MMM

scalars. Appropriate assignment statements are introduced to initialize the scalars corresponding to A and B elements. In addition, assignment statements are introduced before and after the k' loop to initialize the scalars corresponding to C elements, and to write the values back into the array respectively. It is expected that the back-end compiler will allocate floating-point registers for these scalars.

B. Pipeline scheduling

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

- Construct two sequences of length $(M_U \times N_U)$, one containing the multiply operations (we will denote them by mul_1, mul_2, …, $mul_{M_U \times N_U}$) and the other containing the add operations (we will denote them by add_1, add_2, …, $add_{M_U \times N_U}$).
- Interleave the two sequences as shown below to create a single sequence that is obtained by skewing the adds by a factor of L_s, where L_s is an optimization parameter. Intuitively, this interleaving separates most dependent multiplies and adds by $2 \times L_s - 1$ independent instructions to avoid stalling the processor pipeline.

$$\begin{array}{l} mul_1 \\ mul_2 \\ \ldots \\ mul_{L_s} \\ add_1 \\ mul_{L_s+1} \\ add_2 \\ mul_{L_s+2} \\ \ldots \\ mul_{M_U \times N_U - 1} \\ add_{M_U \times N_U - L_s} \\ mul_{M_U \times N_U} \\ add_{M_U \times N_U - L_s + 1} \\ add_{M_U \times N_U - L_s + 2} \\ \ldots \\ add_{M_U \times N_U} \end{array}$$

- Inject the $M_U + N_U$ loads of the elements of A and B into the resulting sequence of arithmetic operations by scheduling a block of I_F (Initial Fetch) loads in the beginning and blocks of N_F loads thereafter as needed. I_F and N_F are optimization parameters.
- Unroll the k'' loop completely. The parameter K_U must be chosen to be large enough to reduce loop overhead, but not so large that the body of the k' loop overflows the L1 instruction cache.
- Reorganize the k' loop to enable the target machine to overlap the loads from one iteration with arithmetic operations from previous iterations. Techniques for accomplishing this are known as software pipelining or modulo scheduling [35].

Note that skewing of dependent adds and multiplies increases register pressure; in particular, the following inequality must hold to avoid register spills (that is, saving in memory the value stored in a processor register):

$$M_U \times N_U + M_U + N_U + L_s \leq N_R \qquad (1)$$

C. Additional details

There are several details we have not discussed so far.

- ATLAS considers a primitive form of L2 cache tiling, driven by a parameter called $CacheEdge$. ATLAS empirically finds the best value of $CacheEdge$ and uses it to compute K_P, based on Inequality 2.

$$2 \times K_P \times N_B + N_B^2 \leq CacheEdge \qquad (2)$$

K_P is further trimmed to be a multiple of N_B. The computed value of K_P is used to block the K dimension of the original problem for one additional level of the memory hierarchy. We will not discuss $CacheEdge$ and K_P in further detail as they are outside the scope of the paper.

- ATLAS chooses the outermost loop order (shown as JIK in Figure 3) during runtime. This technique is known as versioning, because it requires both versions of the code to be compiled in the library.
  The decision of which loop order to choose is based on the size of matrices A and B. If A is smaller than B ($N < M$), ATLAS chooses the JIK loop order. This guarantees that if A fits completely in L2 or higher cache level, it is reused successfully by the loop nest. Similarly, if B is the smaller matrix ($M < N$), ATLAS chooses the IJK loop order.
  For brevity, we consider only the JIK loop order in the rest of the paper.
- Unless the matrices are too small or too large, ATLAS copies tiles of matrices A, B and C to sequential memory to reduce the number of conflict misses and TLB misses during the execution of a mini-MMM. Copying is performed in a manner that allows the copied tiles to be reused by different mini-MMMs. The comments in Figure 3 and the discussion below explain how this goal is achieved for the JIK loop order.
  - Copy all tiles of A before the beginning of the outermost j loop. This is necessary as these tiles are fully reused in *each* iteration of the j loop.
  - Copy all tiles from the j^{th} vertical panel of B before the beginning of the i loop. This is necessary as this panel is fully reused by *each* iteration of the i loop.
  - The single (i, j) tile of C is copied before the beginning of the k loop if $\frac{K_P}{N_B} \geq 12$. This may reduce TLB misses which may be beneficial since this tile is reused by *each* iteration of the k loop, provided that the cost of copying the tile of C to a temporary buffer and back, can be amortized by the computation (large enough K_P).

If the matrices are very small or if there is insufficient memory for copying tiles, the cost of copying might outweigh the benefits of reducing conflict misses during the computation. Therefore, ATLAS generates non-copying versions of mini-MMM as well, and decides at runtime which version to use. Without copying, the number of conflict misses and TLB misses may rise, so it makes sense to use a smaller tile size for the non-copying mini-MMM. In ATLAS, this tile size is another optimization parameter called NCN_B (non-copying N_B). Roughly speaking, the non-copy version is used if (i) the amount of computation is less than some threshold ($M \times N \times K$ in Figure 2 is less than some threshold), and (ii) at least one dimension of one of the three matrices is smaller than $3 \times NCN_B$. The non-copy version is used also when there is insufficient memory to perform the copying.

D. Discussion

Table I lists the optimization parameters for future reference.

Name	Description
N_B	L1 data cache tile size
NCN_B	L1 data cache tile size for non-copying version
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 available, 0 otherwise
F_F, I_F, N_F	Scheduling of loads

TABLE I
SUMMARY OF OPTIMIZATION PARAMETERS

It is intuitively obvious that the performance of the generated mini-MMM code suffers if the values of the optimization parameters in Table I 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. On the other hand, if these parameters are too large, register spills happen. 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. The goal now is to determine optimal values of these parameters for obtaining the best mini-MMM code.

III. Empirical Optimization in ATLAS

ATLAS performs a global search to determine optimal values for the optimization parameters listed in Table I. 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 the 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 micro-benchmarks to determine approximate values for these parameters. It then performs a global search, using the machine parameter values to bound the search space.

A. Estimating machine parameters

The machine parameters measured by ATLAS are the following.

- C_1: the size of L1 data cache.
- N_R: the number of floating-point registers.
- FMA: the availability of a fused multiply-add instruction.
- L_s: although this is not a hardware parameter per se, it is directly related to the latency of floating point multiplication, as explained in Section II-B. ATLAS measures this optimization parameter directly using a micro-benchmark.

The micro-benchmarks used to measure machine parameters are independent of matrix multiplication. For example, the micro-benchmark for estimating C_1 is similar to the one discussed in Hennessy and Patterson [23].

Two other machine parameters are critical for performance: (i) the L1 instruction cache size, and (ii) the number of outstanding loads that the hardware supports. ATLAS does not determine these explicitly using micro-benchmarks; instead, they are considered implicitly during the optimization of matrix multiplication code. For example, the size of the L1 instruction cache limits the K_U parameter in Figure 3. Rather than estimate the size of the instruction cache directly by running a micro-benchmark and using that to determine the amount of unrolling, ATLAS generates a suite of mini-MMM kernels with different K_U values, and selects the kernel that achieves best performance.

B. Global search for optimization parameter values

To find optimal values for the optimization parameters in Table I, 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 1-dimensional optimization problems corresponding to each of the n parameters. When optimizing the value of parameter x_i, it uses reference values for parameters $(x_{i+1}, x_{i+2}, \ldots, x_n)$ that have not yet been optimized. Orthogonal line search is heuristic 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, ..., $k-1$.

The optimization sequence used in ATLAS is the following.

1) Find best N_B.
2) Find best M_U and N_U.
3) Find best K_U.
4) Find best L_s.
5) Find best F_F, I_F, and N_F.
6) Find best NCN_B: a non-copy version of N_B.
7) Find best clean-up codes.

We now discuss each of these steps in greater detail.

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 4 that satisfies the following inequality:

$$16 \leq N_B \leq \min\left(80, \sqrt{C_1}\right) \tag{3}$$

The reference values of M_U and N_U are set to the values closest to each other that satisfy (1). 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 and N_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 inequality (1). Cases when M_U or N_U is 1 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 3 are not checked.

3) Find best K_U: This step is another simple search. Unlike M_U and N_U, K_U does not depend on the number of available registers, so it can be made as large as desired without causing register spills. The main constraint is instruction cache size. ATLAS tries values for K_U between 4 and $\frac{N_B}{2}$ as well as the special values 1 and N_B. The value that gives best performance (based on N_B, M_U and N_U as determined from the previous steps) is declared the optimal value for K_U.

4) Find best L_s: In this step, ATLAS uses L_s values in the interval $[1, 6]$ to schedule the computations in the micro-MMM of Figure 3 to determine the best choice for L_s. It also ensures that the chosen value divides $M_U \times N_U \times K_U$ to facilitate instruction scheduling.

5) Find best F_F, I_F, and N_F: In this step, ATLAS searches for the values of F_F, I_F and N_F. First, ATLAS determines the value of F_F (0 or 1). Then, it searches for the best value of the pair (I_F, N_F) where I_F is in the interval [2,M_U+N_U] and N_F is in the interval [1,M_U+N_U-I_F].

6) Find best NCN_B: For the non-copying version of mini-MMM, ATLAS uses the same values of M_U, N_U, F_F, I_F, and N_F that it uses for the copying version. Without copying, the likelihood of conflict misses is higher, so it makes sense to use a smaller L1 cache tile size than in the version of mini-MMM that performs copying. ATLAS searches for an optimal value of NCN_B in the range $[N_B : -4 : 4]$. We would expect performance to increase initially as the tile size is decreased, but decrease when the tile size becomes too small. ATLAS terminates the search when the performance falls by 20% or more from the best performance it finds during this search. Finally, some restricted searches for better values of K_U and L_s are done.

7) Find best clean-up codes: If the tile size is not a multiple of the original matrix size, there may be left-over rows and columns, at the boundaries of the matrices, forming fractional tiles. To handle these fractional tiles, ATLAS generates clean-up code – a special mini-MMM in which one or more of the dimensions of the three tiles is smaller than N_B. For M and N clean-up only the corresponding dimension is smaller than N_B, while for K cleanup, any of the three dimensions can be smaller than N_B.

For example, ATLAS generates K clean-up codes as follows. For each value of L, representing the size of the K dimension, starting with $L = N_B - 1$ and going down, it generates a specialized version of the mini-MMM code in which some of the loops are fully unrolled. Full unrolling is possible because the shapes of the operands are completely known. When the performance of the general version falls within 1% of the performance of the current specialized version, the generation process is terminated. The current L is declared to be the *Crossover Point*. At runtime, the specialized versions are invoked when the dimension of the left-over tile is greater than L, while the general version is invoked for tile sizes smaller than L.

For M and N clean-up ATLAS produces only a general version, as these are outer loops in the outermost loop nest in Figure 3 and they are not as crucial to performance as K clean-up is. The use of clean-up code in ATLAS is discussed in more detail in [39].

C. Discussion

In optimization problems, there is usually a trade-off between search time and the quality of the solution. For example, we can refine the parameters found by ATLAS by repeating the orthogonal line search some number of times, using the values determined by one search as the reference values for the next search. It is also possible to use more powerful global search algorithms like simulated annealing. However, the potential for obtaining better solutions must be weighed carefully against the increase in installation time. We will address this point in the conclusions.

IV. Model-based Optimization

In this section, we present analytical models for estimating optimal values for the parameters in Table I. To avoid overwhelming the reader, we first present models that ignore interactions between different levels of the memory hierarchy (in this case, L1 data cache and registers). Then, we refine the models to correct for such interactions.

A. Estimating hardware Parameters

Model-based optimization requires more machine parameters than the ATLAS approach because there is no search. The hardware parameters required by our model are as follows.

- C_1, B_1: the capacity and the line size of the L1 data cache.
- C_I: The capacity of the L1 instruction cache.
- $L_\times$: hardware latency of the floating-point multiply instruction
- $|ALU_{FP}|$: number of floating-point functional units
- N_R: the number of floating-point registers.
- FMA: the availability of a fused multiply-add instruction.

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 [44], which accurately measures these values.

B. Estimating N_B

We present our model for estimating N_B using a sequence of refinements for increasingly complex cache organizations. We start with the mini-MMM code in Figure 5, and then adjust the model to take register tiling into account.

The goal is to find the value of N_B that optimizes the use of the L1 data cache. First, we consider a simple cache of capacity C_1, which is fully-associative with optimal replacement policy and unit line-size. There are no conflict misses, and spatial locality is not important.

```
for $j' \in [0:1:N_B-1]$
    for $i' \in [0:1:N_B-1]$
        for $k' \in [0:1:N_B-1]$
            $\mathsf{C}_{i'j'}$ = $\mathsf{C}_{i'j'}$ + $\mathsf{A}_{i'k'} \times \mathsf{B}_{k'j'}$
```

Fig. 5. Schematic Pseudo-Code for mini-MMM

The working set in memory of the mini-MMM loop nest in Figure 5 consists of three $N_B \times N_B$ tiles, one from each of the matrices A, B, and C. For the rest of this section, we will refer to these tiles just as A, B, and C. This working set fits entirely in the cache if Inequality (4) holds.

$$3N_B^2 \leq C_1 \tag{4}$$

A more careful analysis shows that it is not actually necessary for all three $N_B \times N_B$ blocks to reside in the cache for the entire duration of the mini-MMM computation. Consider the mini-MMM code shown in Figure 5. Because k' is the innermost loop, elements of C are computed in succession; once a given element of C has been computed, subsequent iterations of the loop nest do not touch that location again. Therefore, with this loop order, it is sufficient to hold a single element of C in the cache, rather than the entire array. The same reasoning shows that it is sufficient to hold a single column of B in the cache. Putting these facts together, we see that with this loop order, there will be no capacity misses if the cache can hold all of A, a single column of B, and a single element of C. This leads to Inequality (5).

$$N_B^2 + N_B + 1 \leq C_1 \tag{5}$$

1) Correcting for non-unit line size: In reality, caches have non-unit line size. Assume that the line size is B_1. If the three tiles are stored in column major order, both B and C are walked by columns and A is in cache for the entire duration of the mini-MMM. This leads to the refined constraint shown in Inequality (6).

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + \left\lceil \frac{N_B}{B_1} \right\rceil + 1 \leq \frac{C_1}{B_1} \tag{6}$$

2) Correcting for LRU replacement policy: We can further relax the restrictions of our cache organization to allow for Least Recently Used (LRU) replacement instead of optimal replacement. To determine the effects of LRU replacement on the optimal tile size N_B, we must examine the history of memory accesses performed by the loop nest. This analysis is in the spirit of Mattson et.al. [30], who introduced the notions of stack replacement and stack distance.

We start with the innermost loop of the mini-MMM loop nest. A single iteration $\langle j, i, k \rangle$ of this loop touches elements

$$\mathsf{A}_{ik}; \mathsf{B}_{kj}; \mathsf{C}_{ij};$$

where the most recently accessed element is written rightmost in this sequence.

Extending this analysis to the middle loop, we see that the sequence of memory access for a given value of the outer loop indices $\langle j, i \rangle$ is the following (as before, the most recently accessed element is rightmost):

$$\mathsf{A}_{i0}; \mathsf{B}_{0j}; \mathsf{C}_{ij}; \mathsf{A}_{i1}; \mathsf{B}_{1j}; \mathsf{C}_{ij}; \ldots; \mathsf{A}_{i,N_B-1}; \mathsf{B}_{N_B-1,j}; \mathsf{C}_{ij};$$

Note that the location C_{ij} is touched repeatedly, so the corresponding history of memory accesses from least recently accessed to most recently accessed is the following:

$$\mathsf{A}_{i0}; \mathsf{B}_{0j}; \mathsf{A}_{i1}; \mathsf{B}_{1j}; \ldots; \mathsf{A}_{i,N_B-1}; \mathsf{B}_{N_B-1,j}; \mathsf{C}_{ij};$$

Extending this to a single iteration j of the outermost loop, we see that the sequence of memory accesses is the following (in left-to-right, top-to-bottom order):

$$\begin{array}{cccc} \mathsf{A}_{00}; \mathsf{B}_{0j}; & \ldots & \mathsf{A}_{0,N_B-1}; \mathsf{B}_{N_B-1,j}; & \mathsf{C}_{0j}; \\ \mathsf{A}_{10}; \mathsf{B}_{0j}; & \ldots & \mathsf{A}_{1,N_B-1}; \mathsf{B}_{N_B-1,j}; & \mathsf{C}_{1j}; \\ & \vdots & & \\ \mathsf{A}_{N_B-1,0}; \mathsf{B}_{0j}; & \ldots & \mathsf{A}_{N_B-1,N_B-1}; \mathsf{B}_{N_B-1,j}; & \mathsf{C}_{N_B-1,j}; \end{array}$$

Note that the column of B is reused N_B times, and thus the corresponding history of memory accesses from least recently accessed to most recently accessed is

$$\begin{array}{cccc} \mathsf{A}_{00}; & \ldots & \mathsf{A}_{0,N_B-1}; & \mathsf{C}_{0j}; \\ \mathsf{A}_{10}; & \ldots & \mathsf{A}_{1,N_B-1}; & \mathsf{C}_{1j}; \\ & \vdots & & \\ \mathsf{A}_{N_B-1,0}; \mathsf{B}_{0j}; & \ldots & \mathsf{A}_{N_B-1,N_B-1}; \mathsf{B}_{N_B-1,j}; & \mathsf{C}_{N_B-1,j}; \end{array}$$

We do not want to evict the oldest element of this history (A_{00}) because, as we discussed before, A is completely reused in all iterations of the outermost loop. Therefore we need to choose N_B is such a way that this whole history fits in the cache.

Furthermore, after the j^{th} iteration of the outermost loop is complete, the $j+1^{\text{st}}$ iteration will bring in the $j+1^{\text{st}}$ column of B, which participates in an inner product with all the rows of A. Because of LRU, this new column will not be able to "optimally" replace the old j^{th} column of B, since the old column of B has been used quite recently. For the same reason the new element of C, namely $\mathsf{C}_{0,j+1}$, will not be able to optimally replace the old C_{0j}. To account for this, we need extra storage for an extra column of B and an extra element of C.

Putting this all together, we see that if the cache is fully-associative with capacity C_1, line size B_1 and has an LRU replacement policy, we need to cache all of A, two columns of B and a column plus an element of C. This result is expressed formally in Inequality (7).

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B}{B_1} \right\rceil + 1 \leq \frac{C_1}{B_1} \tag{7}$$

Finally, to model the mini-MMM code of Figure 3, which includes register tiling, we need to take into account interactions between the register file and the L1 cache. Thus far, we implicitly assumed that the computation works directly on the scalar elements of the tiles. As Figure 3 shows, the mini-MMM loop nest actually works on register tiles. We refine Inequality (7) by recognizing that considerations of rows, columns, and elements of A, B, and C respectively must be replaced by considerations of horizontal panels, vertical panels, and register tiles instead. Taking this into account, we get Inequality (8).

$$\left\lceil \frac{N_B^2}{B_1} \right\rceil + 3 \left\lceil \frac{N_B \times N_U}{B_1} \right\rceil + \left\lceil \frac{M_U}{B_1} \right\rceil \times N_U \leq \frac{C_1}{B_1} \tag{8}$$

3) Correcting to avoid micro-MMM clean-up code: Note that estimating N_B using Inequality (7), it is possible to get a value for N_B which is not an exact multiple of M_U and N_U. This requires the generation of clean-up code for fractional register tiles at the boundaries of mini-MMM tiles. This complicates code generation, and generally lowers performance. We avoid these complications by trimming the value of N_B determined from Inequality (7) so that it becomes a multiple of M_U and N_U. The ATLAS Code Generator requires N_B to be an even integer, so we enforce this constraint as well.

If N'_B is the tile size obtained by using Inequality (7), we set N_B to the value $\left\lfloor \frac{N'_B}{lcm(M_U,N_U,2)} \right\rfloor \times lcm\,(M_U, N_U, 2)$.

Note this requires that the value of N_B be determined after the values of M_U and N_U have been determined as described below.

4) Other cache organizations: If the cache organization is not fully-associative, conflict misses must be taken into account. Although there is some work in the literature on modeling conflict misses [10, 12], these models are not computationally intractable. Therefore, we do not model conflict misses, although there are some general remarks we can make.

If A, B, and C are copied to $3N_B^2$ contiguous storage locations, Inequality (4) can also be viewed as determining the largest value of N_B for which there are no capacity or conflict misses during the execution of the mini-MMM in *any* cache organization. Although ATLAS usually copies tiles, the code in Figure 3 shows that the three copied tiles are not necessarily adjacent in memory. However, if the set-associativity of the L1 data cache is at least 3, there will be no conflict misses.

Inequality (5) determines the largest N_B for which there are no capacity misses during the execution of the mini-MMM, although there may be conflict misses if the cache is direct-mapped or set-associative. Notice that these conflict misses arise even if data from all three matrix tiles is copied into contiguous memory, because the amount of the data touched by the program is more than the capacity of the cache, and some elements will map to the same cache set.

C. Estimating M_U and N_U

One can look at the register file as a software-controlled, fully-associative cache with unit line size and capacity equal to the number of available registers N_R. Therefore we can use a variant of Inequality (5), to estimate the optimal register file tile size value.

The ATLAS Code Generator uses the KIJ 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 Inequality (5) for registers is Inequality (9), shown below.

$$M_U \times N_U + N_U + 1 \leq N_R \tag{9}$$

Because the register file is software controlled, the ATLAS Code Generator is free to allocate registers differently than Inequality (9) prescribes. In fact, as discussed in Section II, it allocates to registers a $M_U \times 1$ column of A, rather than a single element of A. Furthermore, it needs L_s registers to store temporary values of multiplication operations to schedule for optimal use of the floating point pipelines. Taking into account these details, we refine Inequality (9) to obtain Inequality (10).

$$M_U \times N_U + N_U + M_U + L_s \leq N_R \tag{10}$$

N_R is a hardware parameter, which is measured by the micro-benchmarks. The value of the optimization parameter L_s is estimated as discussed in Section IV-E. Therefore the only unknowns in Inequality (10) are M_U and N_U. We estimate their values using the following procedure.

- Let $M_U = N_U = u$. Solve Inequality (10) for u.
- Let $M_U = \max(u, 1)$. Solve Inequality (10) for N_U.
- Let $N_U = \max(N_U, 1)$
- Let $\langle M_U, N_U \rangle = \langle \max(M_U, N_U), \min(M_U, N_U) \rangle$.

D. Estimating K_U

Although K_U is structurally similar to M_U and N_U, it is obviously not limited by the size of the register file. Therefore the only practical limit for K_U is imposed by the size of the instruction cache. To avoid micro-MMM clean-up code, we trim K_U so that N_B is a multiple of K_U. Note that if $K_U = N_B$ it is left unchanged by this update.

Therefore our model for estimating K_U is to unroll the loop as far as possible within the size constraints of the L1 instruction cache, while ensuring that K_U divides N_B. On most platforms, we found that the loop can be unrolled completely ($K_U = N_B$).

E. Estimating 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.

Studying the description of the scheduling in Section II, 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 $2 \times L_s - 1$ independent instructions will hide the latency of the multiplication. If the floating-point units are fully pipelined and the latency of multiplication is $L_\times$, we get the following inequality, which can be solved to obtain a value for L_s.

$$2 \times L_s - 1 \geq L_\times \tag{11}$$

On some machines, there are multiple floating-point units. If $|ALU_{FP}|$ is the number of floating-point ALUs, Inequality (11) gets refined as follows.

$$\frac{2 \times L_s - 1}{|ALU_{FP}|} \geq L_\times \tag{12}$$

Solving Inequality (12) for L_s, we obtain Inequality (13).

$$L_s = \left\lceil \frac{L_\times \times |ALU_{FP}| + 1}{2} \right\rceil \tag{13}$$

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 (11) holds.

F. Estimating other parameters

Our experience shows that performance is insensitive to the values of F_F, I_F, and N_F optimization parameters. Therefore we set $F_F = 1(true)$, $I_F = 2$ and $N_F = 2$.

FMA is a hardware parameter, independent of the specific application. If our micro-benchmarks determine that the architecture supports a fused multiply-add instruction, we set this parameter appropriately.

Finally, we set $NCN_B = N_B$. That is, we use the same tile size for the non-copying version of mini-MMM as we do for the copying version. In our experiments, ATLAS always decided to use the copying version of mini-MMM[2], so the value of this parameter was moot. A careful model for NCN_B is difficult because it is hard to model conflict misses analytically. There is some work on this in the compiler literature but most of the models are based on counting integer points within certain parameterized polyhedra and appear to be intractable [10, 12]. Fraguela et. al. have proposed another approach to modeling conflict misses when the sizes of matrices are known [16]. In some compilers, this problem is dealt with heuristically by using the *effective* cache capacity, defined to be a fraction (such as $\frac{1}{3}$) of the actual cache capacity, when computing the optimal tile size. In our context, we could set NCN_B to the value determined from Inequality (7) with C_1 replaced with $\frac{C_1}{3}$. We recommend this approach should it become necessary to use a smaller tile size on some architectures.

G. Discussion

We have described a fairly elaborate sequence of models for estimating the optimal value of N_B. In practice, the value found by using Inequality (6), a relatively simple model, is close to the value found by using more elaborate models such as Inequalities (7) and (8).

V. Experimental Results

Models are to be used, not believed.
H. Theil 'Principles of Econometrics'

In this section, we present the results of running ATLAS CGw/s and ATLAS Model on ten common platforms. For all experiments we used the latest stable version of ATLAS, which as of this writing is 3.6.0. Where appropriate, we also present

[2]Using the non-copy version is mainly beneficial when the matrices involved in the computation are either very small or are long and skinny [37].

numbers for ATLAS Unleashed and vendor supported, native BLAS.

We did our experiments on the following platforms.

- RISC, Out-of-order
  - DEC Alpha 21264
  - IBM Power 3
  - IBM Power 4
  - SGI R12K
- RISC In-order
  - Sun UltraSPARC IIIi
  - Intel Itanium2
- CISC, Out-of-order
  - AMD Opteron 240
  - AMD Athlon MP
  - Intel Pentium III
  - Intel Pentium 4

For each platform, we present the following results.

- *Times:*
  - *X-Ray*: time taken by X-Ray to determine hardware parameters.
  - *ATLAS Micro-benchmarks*: time taken by the micro-benchmarks in ATLAS to determine hardware parameters.
  - *ATLAS Optimization Parameter Search*: time taken by global search in ATLAS for determining optimization parameter values.

  We do not report the actual installation time of any of the versions of ATLAS because most of this time is spent in optimizing other BLAS kernels, generating library code, building object modules, etc.

  We do not discuss the timing results in detail as they are not particularly surprising. X-Ray is faster than ATLAS in measuring hardware parameters on nine out of the ten platforms, and has comparable timing (10% slower) on one (IBM Power 3). Moreover, while ATLAS CGw/S spends considerable amount of time, ranging between 8 minutes on the DEC Alpha to more than 8 hours on the Intel Itanium 2, to find optimal values for optimization parameters, the model-based approach takes no measurable time.
- *Performance:*
  - *Optimization parameter values*: values determined by ATLAS CGw/S and ATLAS Model. Where appropriate, we also report these values for ATLAS Unleashed.
  - *mini-MMM performance*: performance of mini-MMM code produced by ATLAS CGw/S, ATLAS Model and ATLAS Unleashed.
  - *MMM performance*: for matrices sized 100×100 to 5000×5000. We report performance of complete MMM computations using (i) vendor supported, native BLAS, and the code produced by (ii) ATLAS CGw/S, (iii) ATLAS Model, (iv) ATLAS Unleashed, and (v) the native Fortran compiler. On each platform, the code produced by ATLAS is compiled with the best C compiler we could find on that platform. The input to the FORTRAN compiler is the standard triply-nested loop shown in Figure 2.

    For vendor supported, native BLAS (labeled "BLAS" on all figures) we used to following libraries and corresponding versions, which were current at the time of our experiments:
    - ∗ DEC Alpha: CXML 5.2
    - ∗ IBM Power 3/4: ESSL 3.3
    - ∗ SGI R12K: SCSL 6.5
    - ∗ SUN UltraSPARC IIIi: Sun One Studio 8
    - ∗ Intel Itanium 2, Pentium III/4: MKL 6.1
    - ∗ AMD Opteron, Athlon: ACML 2.0
- *Sensitivity Analysis:* this describes the relative change of performance as we change one of the optimization parameters, keeping all other parameters fixed to the values found by ATLAS CGw/S. Sensitivity analysis explains how variations in the values of optimization parameters influence the performance of the generated mini-MMM kernel.
  - N_B: change in mini-MMM performance when the value of N_B is changed
  - M_U, N_U: change in mini-MMM performance when values of M_U and N_U are changed. Because optimal values of M_U and N_U depend on the same hardware resource (N_R), we vary them together.
  - K_U: change in min-MMM performance when value of K_U is changed.
  - L_s: change in mini-MMM performance when L_s is changed.
  - F_F, I_F and N_F: we do not show sensitivity graphs for these parameters because performance is relatively insensitive to their values.

A. DEC Alpha 21264

1) mini-MMM: On this machine the model-determined optimization parameters provided performance of about 100 MFLOPS (7%) slower than the ones determined by search. The reason of the difference is the suboptimal selection of the N_B parameter (84 for Atlas Model vs. 72 for ATLAS CGw/S), as can be seen in the N_B sensitivity graph of Figure 12(g).

2) MMM Performance: Figure 12(d) shows the MMM performance.

ATLAS Unleashed produces the fastest BLAS implementation because it uses highly-optimized, hand-tuned BLAS kernels written by Goto. A newer version of these kernels is described in [25]. The native BLAS library is only marginally slower.

Although the gap in performance of the mini-MMM codes produced by ATLAS CGw/S and ATLAS Model is 100 MFLOPS, the gap in performance of complete MMM computations is only about 50 MFLOPS (4%) for large matrices. Finally, we note that the GNU FORTRAN compiler is unable to deliver acceptable performance. We did not have access to the Compaq FORTRAN compiler, so we did not evaluate it.

3) Sensitivity Analysis: Figure 12(e) shows the sensitivity of performance to the values of M_U and N_U. The optimal value is $(4, 4)$, closely followed by $(3, 6)$, and $(6, 3)$. These match our expectations that optimal unroll factors are as close to square as possible, while dividing the tile size $N_B = 72$ without reminder.

Figure 12(f) shows the sensitivity of performance to the value of N_B. Figure 12(g) shows a scaled-up version of this graph in the region of the optimal N_B value. The optimal value for N_B is 88. ATLAS does not find this point because it does not explore tile sizes greater than 80, as explained in Section III, but it chooses a tile size of 72, which is close to optimal. If we use Inequality (8) to determine N_B analytically, we obtain $N_B = 84$. Note that using the simpler model of Inequality (6), we obtain $N_B = 90$, which appears to be almost as good as the value determined by the more complex model.

The N_B sensitivity graph of Figure 12(g) has a saw-tooth of periodicity 4, with notable peaks occurring with a periodicity of 8. The saw-tooth of periodicity 4 arises from the interaction between cache tiling and register tiling - the register tile is $(4, 4)$, so whenever N_B is divisible by 4, there is no clean-up code for fractional register tiles in the mini-MMM code, and performance is good. We do not yet understand why there are notable peaks in the saw-tooth with a periodicity of 8.

Figure 12(h) shows the sensitivity of performance to the value of K_U. On this machine the entire mini-MMM loop body can fit into the L1 instruction cache for values of K_U up to N_B. Performance is relatively insensitive to K_U as long as the value of this parameter is sufficiently large ($K_U > 7$).

Figure 12(i) shows the sensitivity of performance to the value of L_s. The graph is convex upwards, with a peak at 4. The multiplier on this machine has a latency of 4 cycles, so the model for L_s in Section IV, computes $L_s = 5$, which is close to optimal. The inverted-U shape of this graph follows our expectations. For very small values of L_s, dependent multiplications and additions are not well separated and CPU pipeline utilization is low. As L_s grows, the problem gradually disappears, until the performance peak is reached when the full latency of the multiplication is hidden. Increasing L_s further does not improve performance as there is no more latency to hide. On the contrary, more temporary registers are needed to save multiplication results, which causes more register spills to memory, decreasing performance.

B. IBM Power 3

1) mini-MMM: On this machine, mini-MMM code produced by ATLAS Model is about 40 MFLOPS (3%) slower than mini-MMM code produced by ATLAS CGw/S. Figure 13(g) shows that one reason for this difference is the sub-optimal choice of N_B; fixing the values of all parameter other than N_B to the ones chosen by ATLAS CGw/S and using the model-predicted value of 84 for N_B results in mini-MMM code that performs about 100 MFLOPS worse than the mini-MMM code produced by ATLAS CGw/S.

2) MMM Performance: For multiplying large matrices, the handwritten BLAS as well as the codes produced by ATLAS CGw/S, ATLAS Model, and ATLAS Unleashed perform almost identically.

3) Sensitivity Analysis: Figure 13(f) shows the sensitivity of performance to the value of N_B. Figure 13(g) shows a scaled-up version of this graph in the region of the optimal N_B value.

Figure 13(e) shows the sensitivity of performance to the values of M_U and N_U.

Figure 13(h) shows the sensitivity of performance to the value of K_U. On this machine, the entire mini-MMM loop body can fit into the L1 instruction cache for values of K_U up to N_B. Performance is relatively insensitive to K_U as long as the value of this parameter is sufficiently large ($K_U > 5$). We do not understand the sudden drop in performance at $K_U = 3$.

Figure 13(i) shows the sensitivity of performance to the value of L_s. The Power 3 platform has a fused multiply-add instruction, which the ATLAS micro-benchmarks and X-ray find, and the Code Generator exploits, so performance does not depend on the value of L_s.

C. IBM Power 4

1) mini-MMM: On this machine, mini-MMM code produced by ATLAS Model is about 70 MFLOPS (2%) slower than mini-MMM code produced by ATLAS CGw/S. Figure 14(g) shows that one reason for this difference is a slightly sub-optimal choice of N_B; fixing the values of all parameter other than N_B to the ones chosen by ATLAS CGw/S and using the model-predicted value of 56 for N_B results in mini-MMM code that performs slightly worse than the mini-MMM code produced by ATLAS CGw/S.

2) MMM Performance: Figure 14(d) shows MMM performance. For large matrices, the hand-tuned BLAS perform the best, although by a small margin. The code produced by ATLAS Model, ATLAS CGw/S and ATLAS Unleashed perform almost identically. On this machine the native IBM XL Fortran compiler produced relatively good results for small matrices.

3) Sensitivity Analysis: Figure 14(e) shows the sensitivity of performance to changes in the values of M_U and N_U. The parameter values $(4, 4)$ perform best, and these are the values used by both ATLAS CGw/S and ATLAS Model.

Figure 14(f) shows the sensitivity of performance to the value of N_B. Figure 14(g) shows a scaled-up version of this graph in the neighborhood of the N_B value determined by ATLAS CGw/S. Figure 14(f) shows that on this machine, N_B values between 150 and 350 give the best performance of roughly 3.5 GFLOPS. Using Inequality (4) for the L2 cache (capacity of 1.5 MB) gives N_B= 254, while Inequality (8) gives N_B= 436, showing that on this machine, it is better to tile for the L2 cache rather than the L1 cache.

Figure 14(h) shows the sensitivity of performance to the value of K_U. The L1 instruction cache on this machine is large enough that we can set K_U to N_B. As on the Power 3, unrolling by 3 gives poor performance for reasons we do not understand.

Figure 14(i) shows the sensitivity of performance to the value of L_s. The Power 4 platform has a fused multiply-add instruction, which the ATLAS micro-benchmarks find and the Code Generator exploits, so performance does not depend on the value of L_s.

D. SGI R12K

1) mini-MMM: On this machine, mini-MMM code produced by ATLAS Model is about 20 MFLOPS (4%) slower than mini-MMM code produced by ATLAS CGw/S. The performance of both codes is similar to that of mini-MMM code produced by ATLAS Unleashed.

2) MMM Performance: Figure 15(d) shows MMM performance. The hand-coded BLAS perform best by a small margin. On this machine the native compiler (in this case, the SGI MIPSPro) generated relatively good code that was only 20% lower in performance than the hand-coded BLAS, at least for small matrices.

3) Sensitivity Analysis: Figure 15(e) shows the sensitivity of performance to the values of M_U and N_U. This machine has a relatively large number of registers (32), so there is a fairly broad performance plateau in this graph.

Figure 15(f) shows the sensitivity of performance to the value of the N_B. Figure 15(g) shows a scaled-up version of this graph in the region of the optimal N_B value. Figure 15(f) shows that on this machine, N_B values between 300 and 500 give the best performance of roughly 510 MFLOPS. Using Inequality (4) for the L2 cache (capacity of 4MB) gives N_B= 418, while Inequality (8) gives $N_B = 718$, showing that on this machine, it is better to tile for the L2 cache rather than the L1 cache.

Figure 15(h) shows the sensitivity of performance to the value of the K_U. On this machine, the instruction cache is large enough that full unrolling (K_U=N_B) is possible.

Figure 15(i) shows the sensitivity of performance to the value of the L_s. The R12K processor has a fused multiply-add instruction, so we would expect performance of the generated code to be insensitive to the value of L_s. While this is borne out by Figure 15(i), notice that Table 15(b) shows that the micro-benchmark used by ATLAS did not discover the fused multiply-add instruction on this machine ($FMA = 0$)! It is worth mentioning that on this platform the FMA instruction, while present in the ISA, is not backed up by a real FMA pipeline in hardware. Instead it allows the two separate functional units (for multiplication and addition respectively) to be used sequentially saving one latency cycle. Therefore, in theory, peak performance is achievable even by using separate multiply and add instructions. Although ATLAS Code Generator schedules code using $L_s = 3$, the SGI MIPSPro compiler is clever enough to discover the separated multiplies and adds, and fuse them. In fact the compiler is able to do this even when $L_s = 20$, which is impressive.

E. Sun UltraSPARC IIIi

1) mini-MMM: On this machine, mini-MMM code produced by ATLAS Model is about 160 MFLOPS (17%) *faster* than mini-MMM code produced by ATLAS CGw/S. The main reason for this is that the micro-benchmarks used by ATLAS incorrectly measured the capacity of the L1 data cache as 16 KB, rather than 64 KB. Therefore ATLAS only searched for N_B values less than 44. Our micro-benchmarks on the other hand correctly measured the capacity of the L1 cache, so the model estimated $N_B = 84$, which gave better performance as can be seen in Figure 16(g).

2) MMM Performance: Figure 16(d) shows the MMM performance. On this machine, the hand-coded BLAS and ATLAS Unleashed performed roughly 50% better than the code produced by ATLAS CGw/S. The reason for this difference is that the mini-MMM code in ATLAS Unleashed (and perhaps the hand-coded BLAS) pre-fetches portions of the A and B matrices required for the next mini-MMM. This may be related to the Level-3 pre-fetching idea of Gustavson et. al. [3].

3) Sensitivity Analysis: Figure 16(e) shows the sensitivity of performance to the values of M_U and N_U.

Figure 16(f) shows the sensitivity of performance to the value of the N_B. Figure 16(g) shows a scaled-up version of this graph in the region of the optimal N_B value. On this machine, as on many other machines, it is better to tile for the L2 cache, as can be seen in Figure 16(f). Using Inequality (4) for the L2 cache (capacity of 1 MB), we obtain $N_B = 208$, which gives roughly 1380 MFLOPS. Using Inequality (8), we obtain $N_B = 356$, which is close to the N_B value in Figure 16(f) where the performance drops rapidly.

Figure 16(h) shows the sensitivity of performance to the value of the K_U. On this machine, the instruction cache is large enough that full unrolling (K_U=N_B) is possible.

Figure 16(i) shows the sensitivity of performance to the value of the L_s. This machine does not have a fused multiply-add instruction, so the value of the L_s parameter affects performance. Both the model and ATLAS CGw/S find good values for this parameter.

F. Intel Itanium 2

1) mini-MMM: 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 examine the sensitivity graphs to understand the reasons why ATLAS Model is doing so poorly.

Figure 17(g) shows that one reason for this difference is that ATLAS Model used $N_B = 30$, whereas ATLAS CGw/S used $N_B = 80$. ATLAS CGw/S uses $N_B = 80$ because it disregards the L1 data cache size (16KB) and considers directly the L2 cache size (256KB), and therefore the expression $\min\left(80, \sqrt{C}\right)$ in Inequality (3) evaluates to 80, the largest possible value of N_B in the search space used by ATLAS.

While the value $N_B = 30$ used by ATLAS Model is correct with respect to the L1 data cache size, Intel Itanium 2 does not allow storing floating point numbers in the L1 data cache, and thus L2 has to be considered instead. Once we incorporate in X-Ray the ability to measure this specific hardware feature, the shortcoming of ATLAS Model will be resolved.

2) MMM Performance: Figure 17(d) shows MMM performance. The hand-written BLAS and ATLAS Unleashed give the best performance. The code produced by ATLAS CGw/S runs about 1.5 GFlops slower than the hand-written BLAS, while the code produced by ATLAS Model runs about 3.5 GFlops slower.

3) Sensitivity Analysis: Figure 17(e) shows the sensitivity of performance to the values of M_U and N_U. The Itanium has

128 general-purpose registers, so the optimal register tiles are relatively large. There is a broad plateau of (M_U,N_U) values that give excellent performance.

Figure 17(f) shows the sensitivity of performance to the value of the N_B. Figure 17(g) shows a scaled-up version of this graph in the region of the optimal N_B value. Figure 17(f) shows that on this machine, the best performance is obtained by tiling for the L3 cache! Indeed, using Inequality (4) for the L3 cache (capacity of 3 MB), we obtain $N_B = 360$, which gives roughly 4.6 GFLOPS. Figure 17(f) shows that this value is close to optimal. Using Inequality (8), we obtain $N_B = 610$, which is close to the N_B value in Figure 17(f) where the performance starts to drop.

Figure 17(h) shows the sensitivity of performance to the value of K_U. On the Itanium, unlike on other machines in our study, performance is highly sensitive to the value of K_U. The main reason is the large register tile $(M_U, N_U) = (10, 10)$; after unrolling the micro-MMM loops, we get a very long straight-line code sequence. Furthermore, unrolling of the k'' loop creates numerous copies of this code sequence. Unfortunately, the L1 instruction cache on this machine has a capacity of 32 KB, so it can hold only about 9 copies of the micro-MMM code sequence. Therefore, performance drops off dramatically for values of K_U greater than 9 or 10.

Since this is the only machine in our study in which the K_U parameter mattered, we decided to investigate the sensitivity graph more carefully. Figure 6 shows a magnified version of Figure 17(h) in the interval $K_U \in [0, 15]$. We would expect the K_U sensitivity graph to exhibit the typical inverted-U shape, and it more or less does. However, performance for $K_U = 7$ is significantly worse than the performance for $K_U = 6$, and $K_U = 8$, which appears anomalous.

The anomaly arises from clean-up code that is required when K_U does not divide N_B evenly (see the k' loop in the tiled code in Figure 3). If we unroll the k' loop by K_U, the number of times the completely unrolled micro-MMM code is replicated inside the mini-MMM is not K_U, but $K_U + N_B\%K_U$ (% is the reminder from integer division). The first term in the sum is the expected number of repetitions inside the unrolled k' loop, while the second part is the clean-up code which takes care of the case when K_U does not divide N_B exactly. This second piece of code is still part of the mini-MMM loop nest, and it has to be stored in the L1 instruction cache during execution to achieve optimal performance.

For $N_B = 80$, performance increases initially as K_U increases because loop overhead is reduced. When $K_U = 6$, there are 8 copies of the unrolled micro-MMM code in the mini-MMM, and this is close to the I-cache limit. When $K_U = 7$, there are $7 + 80\%7 = 10$ copies of the micro-MMM code, which exceeds the I-cache limit, and performance drops substantially. However, when $K_U = 8$, there is no clean-up code, and there are only 8 copies of the unrolled micro-MMM code, so performance goes up again. Beyond this point, the code sizes overflows the I-cache and grows larger, and performance degrades gradually. Ultimately, performance is limited by the rate at which L1 I-cache misses can be serviced. For $N_B = 360$, the trends are similar, but the effect of clean-up code is less because the clean-up code performs a smaller fraction of the computations of the k' loop (less than 1% compared to about 5% for $N_B = 80$).

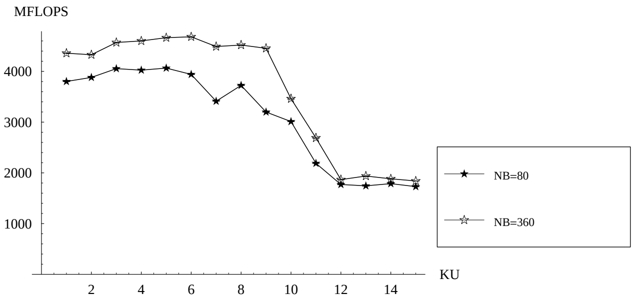

Fig. 6. Intel Itanium 2: Sensitivity of performance to K_U

Figure 17(i) shows the sensitivity of performance to the value of the L_s. The Itanium has a fused multiply-add instruction, so performance is insensitive to the L_s parameter.

In summary, the code produced by ATLAS Model on this machine did not perform as well as the code produced by ATLAS CGw/S. However, this is because ATLAS Model tiled for the L1 cache, whereas on this machine, the best performance is obtained by tiling for L3 cache. ATLAS CGw/S gets better performance because the tile size is set to a larger value than the value used by ATLAS Model.

G. AMD Opteron 240

1) mini-MMM: Table 18(c) shows that on this machine, the mini-MMM code generated by ATLAS Model runs roughly 38% slower than the code generated by ATLAS CGw/S. The values of almost all optimization parameters determined by the two systems are different, so it is not obvious where the problem is. To get some insight, it is necessary to look at the sensitivity graphs.

Figure 18(f) shows the performance sensitivity graph for N_B. Both 60 and 88 appear to be reasonable values, so the problem with ATLAS Model is not in its choice of N_B. Because K_U is bound to the value of N_B, the only remaining differences are those between M_U, N_U, L_s, and FMA. Table 18(b) shows that ATLAS Model chose $M_U = 2$, $N_U = 1$, $FMA = 0$, while ATLAS CGw/S chose $M_U = 6$, $N_U = 1$, $FMA = 1$. We verified experimentally that if the model had chosen $M_U = 6$ and $FMA = 1$, keeping the rest of the parameters the same, the mini-MMM performance becomes 2050 MFLOPS, closing the performance gap with ATLAS CGw/S.

The parameters values used by ATLAS CGw/S are puzzling for several reasons. First, the Opteron does not have an FMA instruction, so it is not clear why ATLAS CGw/S chose to set $FMA = 1$. Second, choosing 6 and 1 for the values of M_U and N_U violates Inequality (10) since the Opteron has only 8 registers.

We address the problem of the register-tile size first. Recall that Inequality (10) stems from the fact that ATLAS uses registers to multiply an $M_U \times 1$ vector-tile of matrix A (which we call $\bar{a}$) with a $1 \times N_U$ vector-tile of matrix B (which we call $\bar{b}$), accumulating the result into an $M_U \times N_U$ tile of matrix C (which we call $\bar{c}$). Notice that if $N_U = 1$, then $\bar{b}$ is a single scalar that is multiplied by each element of $\bar{a}$. Therefore *no reuse exists* for elements of $\bar{a}$. This observation lets us generate the code in Figure 7, which 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 $\bar{a}$.

```
$rc_1 \leftarrow \bar{c}_1 \ldots rc_6 \leftarrow \bar{c}_6$
...
loop $k$
{
    $rb \leftarrow \bar{b}_1$

    $rt \leftarrow \bar{a}_1$
    $rt \leftarrow rt \times rb$
    $rc_1 \leftarrow rc_1 + rt$

    $rt \leftarrow \bar{a}_2$
    $rt \leftarrow rt \times rb$
    $rc_2 \leftarrow rc_2 + rt$

    $\vdots$

    $rt \leftarrow \bar{a}_6$
    $rt \leftarrow rt \times rb$
    $rc_6 \leftarrow rc_6 + rt$
}
...
$\bar{c}_1 \leftarrow rc_1 \ldots \bar{c}_6 \leftarrow rc_6$
```

Fig. 7. $(M_U, N_U) = (6, 1)$ code for x86 CISC

Even if there are enough logical registers, this kind of scheduling may be beneficial if the ISA is 2-address rather than 3-address, because one of the operands is overwritten. This is true on the Opteron when the 16 SSE vector registers are used to hold scalar values, which is GCC's default behavior. Even though Inequality 1 prescribes 3×3 register tiles, the refined model prescribes 14×1 tiles. Experiments show that this performs better [38].

One might expect that this code will not perform well because there are dependences between most of the instructions that arise from the use of temporary register rt. In fact, experiments show that the code in Figure 7 performs well because of two architectural features of the Opteron.

1) *Out-of-order execution*: it is possible to schedule several multiplications in successive cycles without waiting for the first one to complete.
2) *Register renaming*: the single temporary register rt is renamed to a different physical register for each pair of multiply-add instructions.

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

These insights permit us to refine the model described in Section IV as follows: for processors with out-of-order execution and a small number of logical registers, set $N_U = 1$, $M_U = N_R - 2$, $FMA = 1$.

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 8. 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 7.

```
$rc_1 \leftarrow \bar{c}_1 \ldots rc_6 \leftarrow \bar{c}_6$
...
loop $k$
{
    $ra_1 \leftarrow \bar{a}_1$
    $rb \leftarrow \bar{b}_1$
    $rc_1 \leftarrow rc_1 + ra_1 \times rb$
    $ra_2 \leftarrow \bar{a}_2$
    $ra_3 \leftarrow \bar{a}_3$
    $rc_2 \leftarrow rc_2 + ra_2 \times rb$
    $rc_3 \leftarrow rc_3 + ra_3 \times rb$
    $ra_4 \leftarrow \bar{a}_4$
    $ra_5 \leftarrow \bar{a}_5$
    $rc_4 \leftarrow rc_4 + ra_4 \times rb$
    $rc_5 \leftarrow rc_5 + ra_5 \times rb$
    $ra_6 \leftarrow \bar{a}_6$
    $rc_6 \leftarrow rc_6 + ra_6 \times rb$
}
...
$\bar{c}_1 \leftarrow rc_1 \ldots \bar{c}_6 \leftarrow rc_6$
```

Fig. 8. ATLAS unroll $(M_U, N_U) = (6, 1)$ code for x86 CISC

Notice that the ATLAS Code Generator itself is not aware that the code of Figure 7 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.

2) MMM Performance: Figure 18(d) shows the MMM performance. ATLAS Unleashed is once again the fastest implementation here, as it uses the highly-optimized, hand-tuned BLAS kernels, using the SSE2 SIMD instructions, for which the ATLAS Code Generator does not generate code. The native BLAS library is about 200 MFLOPS slower on average. ATLAS CGw/S and ATLAS Model perform at the same level as their corresponding mini-MMM kernels.

Refining the model as explained above brings ATLAS Model on par with ATLAS CGw/s. To bridge the gap between ATLAS CGw/S and user contributed code, we would need a different code generator – one that understands SIMD and prefetch instructions. GCC exposes these as intrinsic functions and we plan to explore this in our future work.

3) Performance Sensitivity Analysis: Figure 18(f) shows the sensitivity of performance to the value of the N_B optimization parameter. The first drop in performance is the result of L1 data cache misses starting to occur. This fact is accurately captured by our model for N_B in Inequality (8). Solving the inequality for $C = 8192$ (the L1 data cache capacity in double-sized floating-point values), we obtain $N_B = 89$. Similarly the second drop in performance in Figure 18(f) can be explained by applying the same model to the 1MB L2 cache.

Figure 18(e) shows the performance sensitivity to the values of the M_U and N_U optimization parameters. As discussed in Section V-G.1, the optimal value, is $(6, 1)$. From the graph we can see that the only plausible values are those with $N_U = 1$. Furthermore, performance increases while we grow M_U from 1 to 6, while it suddenly drops for $M_U = 7$. This is easily explained by our refined model, as $M_U + 2 \leq N_R$ would require 9 registers, while only 8 are available.

Figure 18(h) shows the performance sensitivity to the value of the K_U optimization parameter. On this machine the entire mini-MMM loop body can fit into the L1 instruction cache for arbitrary K_U values (up to $K_U = N_B$). Performance is relatively insensitive to K_U as long as this unroll factor is sufficiently large ($K_U > 10$).

Figure 18(i) shows the performance sensitivity to the value of the L_s optimization parameter. As we mentioned before, when $FMA = 1$, the L_s optimization parameter does not influence the generated code. Therefore, performance is constant with respect to L_s.

H. AMD Athlon MP

The AMD Athlon implements the x86 instruction set, so we would expect the experimental results to be similar to those on the Opteron.

1) mini-MMM: Table 19(c) shows that on this machine, the mini-MMM code generated by ATLAS Model runs roughly 20% slower than the code generated by ATLAS CGw/S. Figure 19(f) shows that the choice of N_B made by the model is reasonable, while Figure 19(e) shows that the register-tile values were not chosen optimally by the model, as on the Opteron. The problem and its solution are similar to those on the Opteron.

2) MMM Performance: Figure 19(d) shows MMM performance. ATLAS Unleashed out-performs the other approaches by a significant margin. The hand-coded BLAS do almost as well, followed by ATLAS CGw/S.

3) Sensitivity Analysis: Figure 19(e) shows the sensitivity of performance to the values of M_U and N_U.

Figure 19(f) shows the sensitivity of performance to the value of N_B. Figure 19(g) shows a scaled-up version of this graph in the region of the optimal N_B value. Both ATLAS Model and ATLAS CGw/S choose good values of N_B. In Figure 19(g), the saw-tooth with period 2 arises from the overhead of executing clean-up code when the value of N_B is odd, and therefore not divisible by the value of $M_U (= 2)$. As on other machines, we do not understand the saw-tooth with period 4 that has larger spikes in performance.

Figure 19(h) shows the sensitivity of performance to the value of K_U. The L1 I-cache is large enough to permit full unrolling ($K_U = N_B$). However, the sensitivity graph of K_U is anomalous; performance is relatively low for all values of K_U other than $K_U = N_B$. By examining the code produced by the native compiler (GCC), we found that this anomaly arose from interference between instruction scheduling in ATLAS and instruction scheduling in GCC. Notice that ATLAS CGw/S uses $FMA = 0$, so it attempts to schedule instructions and perform software pipelining in the mini-MMM code. Fully unrolling the k' loop ($K_U = N_B$) produces straight-line code which is easier for GCC to schedule.

To verify this conjecture, we redid the K_U sensitivity study with FMA set to 1. Figure 9 shows the results. Setting $FMA = 1$ dissuades the ATLAS Code Generator from attempting to schedule code, so GCC has an easier job, producing a K_U sensitivity graph that is in line with what we would expect.

Notice that our refined model, described in the context of the Opteron, does exactly on this. Using this model, mini-MMM performance is 1544 MFLOPS, which is faster than the performance of the mini-MMM produced by ATLAS CGw/S.

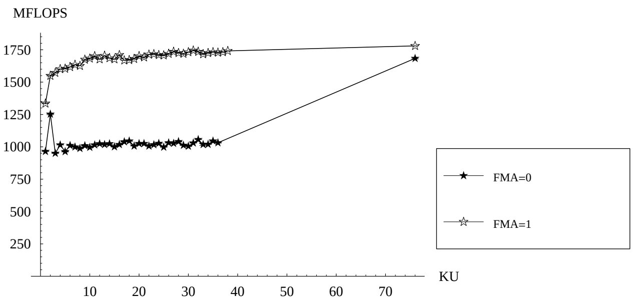

Fig. 9. AMD Athlon MP: Sensitivity of performance to K_U

Figure 19(i) shows the sensitivity of performance to the value of the L_s.

I. Pentium III

1) mini-MMM: On this machine, mini-MMM code produced by ATLAS Model is about 50 MFLOPS (6%) slower than mini-MMM code produced by ATLAS CGw/S. The code produced by ATLAS Unleashed performs roughly 50 MFLOPS better than the code produced by ATLAS CGw/S.

The difference in performance between the codes produced by ATLAS CGw/S and ATLAS Model arises mostly from the sub-optimal register tile chosen by the model, as explained in the context of the Opteron in Section V-G. Using $(6, 1)$ as the register tile raises mini-MMM performance to 916 MFLOPS.

2) MMM Performance: Figure 20(d) shows MMM performance. The hand-coded BLAS perform at roughly 1100 MFLOPS, whereas the codes produced by ATLAS CGw/S and ATLAS Unleashed perform roughly at 900 MFLOPS. The code produced by ATLAS Model runs roughly at 850 MFLOPS; using the refined model improves performance to a point that is slightly above the performance of code produced by ATLAS CGw/S.

3) Sensitivity Analysis: Figure 20(e) shows the sensitivity of performance to the values of M_U and N_U. Like all x86 machines, the Pentium III has a limited number of logical registers. Our base-line model picked $(2, 1)$ for the register tile, whereas ATLAS CGw/S chose $(4, 1)$. If we use the refined model described in Section V-G, the size of the register tile becomes $(6, 1)$, and mini-MMM performance rises to 916 MFLOPS.

Figure 20(f) shows the sensitivity of performance to the value of N_B. Figure 20(g) shows a scaled-up version of this graph in the region of the optimal N_B value. The broad peak in Figure 20(f) arises from the influence of the L2 cache (capacity of 512 KB). Using Inequality (4) for the L2 cache, we obtain $N_B = 104$, which is the N_B values where the peak starts, while Inequality (8) gives $N_B = 164$, which corresponds to the N_B value where the peak ends. The L2 cache on the Pentium III is 8-way set-associative, so the drop in performance between $N_B = 104$ and $N_B = 164$ is small.

Figure 20(h) shows the sensitivity of performance to the value of the K_U. On this machine, the L1 instruction cache is large enough to permit full unrolling ($K_U = N_B$).

Figure 20(i) shows the sensitivity of performance to the value of the L_s. There is no fused multiply-add instruction, so performance is sensitive to the value of L_s, but both ATLAS Model and ATLAS CGw/S find reasonable values for this parameter. If we use the refined model described in Section V-G, we set $FMA = 1$, and the value of the L_s parameter becomes irrelevant.

J. Pentium 4

1) mini-MMM: On this machine, mini-MMM code produced by ATLAS Model is about 600 MFLOPS (40%) slower than mini-MMM code produced by ATLAS CGw/S. This is mostly because of the sub-optimal register tile used by ATLAS Model; changing it to $(6, 1)$ improves the performance of mini-MMM code produced by ATLAS Model to 1445 MFLOPS, which is only 50 MFLOPS (3%) less than the performance of the mini-MMM code produced by ATLAS CGw/S.

The mini-MMM produced by ATLAS Unleashed is roughly twice as fast as the mini-MMM produced by ATLAS Model because this code uses the SSE2 vector extensions to the x86 instruction set.

2) MMM Performance: Figure 21(d) shows the MMM performance. The hand-coded BLAS routines for this machine perform best, followed by the code produced by ATLAS Unleashed. Both the hand-coded BLAS and the code produced by ATLAS Unleashed use the SSE2 vector extensions, and this accounts for most of the gap between these codes and the codes produced by ATLAS Model and ATLAS CGw/S. We do not know why the hand-coded BLAS perform substantially better than the code produced by ATLAS Unleashed.

The gap in performance between the codes produced by ATLAS CGw/S and ATLAS Model disappears if the refined model for register tiles is used.

3) Sensitivity Analysis: Figure 21(e) shows the sensitivity of performance to the values of M_U and N_U. This figure shows that the best register tile is $(5, 1)$, which produces mini-MMM code that runs at 1605 MFLOPS. Using $(6, 1)$ as the register tile is not as good because it reduces performance to 1521 MFLOPS.

Figure 21(f) shows the sensitivity of performance to the value of the N_B. Figure 21(g) shows a scaled-up version of this graph in the region of the optimal N_B value. Both ATLAS Model and ATLAS CGw/S choose good tile sizes for the L1 cache. Tiling for the L2 cache gives slightly better performance. The L2 cache on this machine has a capacity of 256 KB; using Inequalities (4) and (8), we get $N_B = 105$ and $N_B = 180$, which agree well with the data.

Figure 21(h) shows the sensitivity of performance to the value of K_U. On this machine, the L1 instruction cache is large enough to permit full unrolling ($K_U = N_B$).

Figure 21(i) shows the sensitivity of performance to the value of L_s.

K. Discussion

The experimental results in this section can be summarized as follows. Figure 10 describes the analytical models used to compute values for the optimization parameters. This figure also shows the refined model used to compute register tile values for the x86 architectures.

Figure 11 shows the relative performance of the mini-MMM codes produced by ATLAS Model and by ATLAS Unleashed, using the performance of the codes produced by ATLAS CGw/S as the base line (the 100% line in this figure represents the performance of ATLAS CGw/S on all machines). All the performance numbers for ATLAS Model in this graph are obtained by tiling for the L1 cache.

We see that on all machines other than the Itanium, the codes produced by using the analytical models perform almost as well or slightly better than the codes produced using global search. On the Itanium, we saw that it is best to tile for the L3 cache, rather than the L1 cache. By using the L2 cache instead, ATLAS CGw/S was able to obtain some of the benefits of tiling for the L3 cache. If we use this value in the model of Figure 10, we produce mini-MMM code of comparable performance. Using the actual capacity of the L3 cache gives even better performance.

In our experiments we noticed that on several platforms, we get better MMM performance by tiling for a lower cache level, such as L2 or L3, rather than L1. This may result in a large value for N_B, which may hurt overall performance if the resulting MMM library routine is invoked from other routines such as LU and Cholesky factorizations [22]. It is unclear to us that this is an issue in the context of compilers, where codes like LU and Cholesky would be optimized directly, rather than built upon MMM.

- **Estimating FMA:**
  Use the machine parameter FMA
- **Estimating L_s:**

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

- **Estimating M_U and N_U:**

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

  1) $M_U, N_U \leftarrow u$.
  2) Solve constraint for u.
  3) $M_U \leftarrow \max(u, 1)$.
  4) Solve constraint for N_U.
  5) $N_U \leftarrow \max(N_U, 1)$.
  6) If $M_U < N_U$ then swap M_U and N_U.
  7) **Refined Model:** If $N_U = 1$ then
     - $M_U \leftarrow N_R - 2$
     - $N_U \leftarrow 1$
     - $FMA \leftarrow 1$
- **Estimating N_B:**

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

  Trim N_B, to make it a multiple of M_U, N_U, and 2.
- **Estimating K_U:**
  Choose K_U as the maximum value for which mini-MMM fits in the L1 instruction cache. Trim K_U to make it divide N_B evenly.
- **Estimating F_F, I_F, and N_F:**

$$F_F = 0, I_F = 2, N_F = 2$$

Fig. 10. Summary of Model

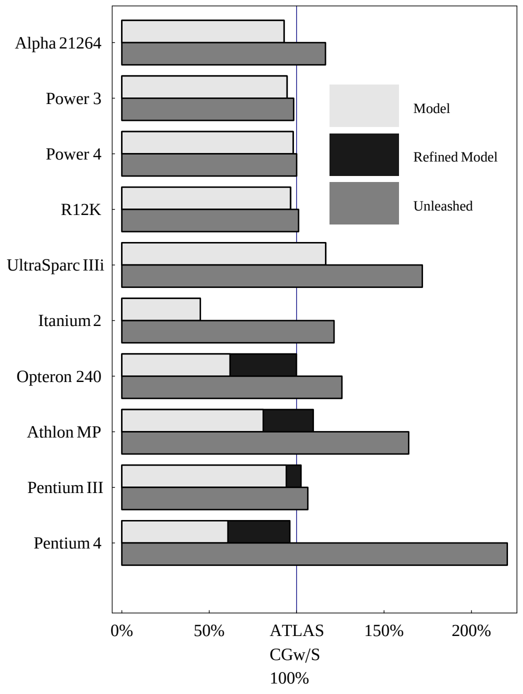

Fig. 11. Summary of mini-MMM Performance. Performance numbers are normalized to that of ATLAS CGw/S, which is presented as 100%

VI. Conclusions and Future Work

. . . the end of all our exploring
Will be to arrive where we started
And know the place for the first time.
T.S.Eliot, Four Quartets

The experimental results in this paper demonstrate that it is possible to use analytical models to determine near-optimal values for the optimization parameters needed in the ATLAS system to produce high-quality BLAS codes. The models in this paper were designed to be compatible with the ATLAS Code Generator; for example, since ATLAS uses square cache tiles, we had only one parameter N_B, whereas a different Code Generator that uses general rectangular tiles may require three cache tile parameters. Van de Geijn and co-workers have considered such models in their work on optimizing matrix multiplication code for multi-level memory hierarchies [20, 21, 24].

Our results show that using models to determine values for the optimization parameters is much faster than using empirical search. However, this does not imply that search has no role to play in the generation of high-performance code. Systems like FFTW and SPIRAL use search not to choose optimal values for transformation parameters, but to choose an optimal algorithm from a whole suite of algorithms. We do not know if model-driven optimization is effective in this context. Even in the relatively simple context of the BLAS, there are aspects of program behavior that may not be worth modeling in practice even if they can be modeled in principle. For example, the analytical models for N_B described in Section IV ignore conflict misses. Although there is some work in the compiler literature on modeling conflict misses [10, 12], these models appear to be computationally intractable. Fortunately, the effect of conflict misses on performance can be reduced by appropriate copying. If necessary, the value of N_B found by the model can be refined by local search in the neighborhood of the N_B value predicted by the model. This combination of modeling and local search may be the most tractable approach for optimizing large programs for complex high-performance architectures.

At the end of this paper, we are left with the same question that we asked at its beginning: how do we improve the state of the art of compilers? Conventional wisdom holds that current compilers are unable to produce high-quality code because the analytical models they use to estimate optimization parameter values are overly simplistic compared to the complexity of modern high-performance architectures. The results in this paper contradict this conventional wisdom, and suggest that there is no intrinsic reason why compilers cannot use analytical models to generate excellent code, at least for the BLAS.

However, it is important not to underestimate the challenge in improving general-purpose compilers to bridge the current performance gap with library generators. Although the techniques used by ATLAS, such as loop tiling, unrolling, and instruction scheduling, have been in the compiler literature for many years, it is not easy to incorporate them into general-purpose compilers. For example, transformations such as tiling are not always legal, so a general-purpose compiler must perform dependence analysis before transforming a program. In contrast, the implementor of a library generator focuses on one application and knows the precise structure of the code to be generated for that application, so he is not encumbered by the baggage required to support restructuring of general codes. At the very least, improving the state of the art of compilation technology will require an open compiler infrastructure which permits researchers to experiment easily with different transformations and to vary the parameters of those transformations. This has been a long-standing problem, and no adequate infrastructure exists in spite of many attempts.

An equally important conclusion of this study is that there is still a significant gap in performance between the code generated by ATLAS CGw/S and the vendor BLAS routines. Although we understand some of the reasons for this gap, the problem of automating library generation remains open. The high cost of library and application tuning makes this one of the most important questions we face today.

REFERENCES

[1] ATLAS homepage. http://math-atlas.sourceforge.net/.

[2] The PHiPAC home page. `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[3] R. C. Agarwal, F. G. Gustavson, and M. Zubair. Improving performance of linear algebra algorithms for dense matrices using algorithmic prefetch. *IBM Journal of Research and Development*, 38(3):265–275, 1994.

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

[5] Uptal Banerjee. Unimodular transformations of double loops. In *Languages and compilers for parallel computing*, pages 192–219, 1990.

[6] Jeff Bilmes, Krste Asanović, Chee whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[7] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? In *INTEGRATION, the VLSI Journal*, volume 17, pages 33–51. 1994.

[8] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1988.

[9] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 53–65, 1990.

[10] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 286–297. ACM Press, 2001.

[11] Michael Cierniak and Wei Li. Unifying data and control transformations for distributed shared memory machines. In *SIGPLAN 1995 conference on Programming Languages Design and Implementation*, June 1995.

[12] Phillipe Claus. Counting solutions to linear and nonlinear constraints through Erhart polynomials. In *ACM International Conference on Supercomputing*. ACM, May 1996.

[13] 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.

[14] Paul Feautrier. Some efficient solutions to the affine scheduling problem - part 1: one dimensional time. *International Journal of Parallel Programming*, October 1992.

[15] Martin Fowler. Yet another optimization article. *IEEE Software*, pages 20–21, May/June 2002.

[16] B. B. Fraguela, R. Doallo, and E. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 221–231, 1999.

[17] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[18] 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”.

[19] Stefan Goedecker and Adolfy Hoisie. *Performance Optimization of Numerically Intensive Codes*. Society for Industrial & Applied Mathematics, 2001.

[20] Kazushige Goto and Robert van de Geijn. On reducing tlb misses in matrix multiplication. Technical Report TR-2002-55, University of Texas at Austin, Department of Computer Sciences, November 2002.

[21] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix algorithms. In *Proceedings of International Conference of Computational Science - ICCS 2001: San Francisco, CA, USA, May 28-30, 2001 Proceedings, Part I*, pages 51–60. Springer, 2001.

[22] Fred Gustavson. Personal communication.

[23] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[24] G. Henry. Flexible high-performance matrix multiply via self-modifying runtime code, 2001.

[25] High-performance blas by kazushige goto. `http://www.cs.utexas.edu/users/flame/goto/`.

[26] Jeremy Johnson, Robert W. Johnson, David A. Padua, and Jianxin Xiong. Searching for the best FFT formulas with the SPL compiler. In *Proc. of the 13th International Workshop on Languages and Compilers for Parallel Computing*, pages 109–124, 2000.

[27] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Programming Languages, Design and Implementation*. ACM SIGPLAN, June 1997.

[28] Induprakas Kodukula and Keshav Pingali. Imperfectly nested loop transformations for memory hierarchy management. In *International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[29] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 1993.

[30] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–92, 1970.

[31] A. C. McKellar and E. G. Coffman, Jr. Organizing matrices and matrix operations for paged memory systems. *Commun. ACM*, 12(3):153–165, 1969.

[32] David Padua and Michael Wolfe. Advanced compiler optimization for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[33] 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”.

[34] Joan McComb Ramesh C. Agarwal, Fred G. Gustavson and Stanley Schmidt. Engineering and Scientific Subroutine Library Release 3 for IBM ES/3090 Vector Multiprocessors. *IBM Systems Journal*, 28(2):345–350, 1989.

[35] B. Ramakrishna Rau. Iterative modulo scheduling. Technical Report HPL-94-115, Hewlett-Packard Research Laboratories, November 1995.

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

[37] R. Clint Whaley. Personal communication.

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

[39] R. Clint Whaley. User contribution to atlas. `http://math-atlas.sourceforge.net/devel/atlas_contrib`.

[40] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. Accepted for publication in *Software: Practice and Experience*, 2004. `http://www.cs.utk.edu/~rwhaley/papers/spercw04.ps`.

[41] 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. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[42] Michael E. Wolf and Monica S. Lam. An algorithmic approach to compund loop transformations. In *Advances in Languages and Compilers for Parallel Computing*. Pitman Publisher, 1991.

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

[44] Kamen Yotov, Keshav Pingali, and Paul Stodghill. X-ray: A tool for automatic measurement of architectural parameters. Technical Report TR2004-1966, Cornell University, Computer Science, October 2004.

Feature	Value
Architecture	Out-Of-Order, RISC
CPU Core Frequency	833 MHz
L1 Data Cache	64 KB, 64 B/line, 2-way
L1 Instruction Cache	64 KB, 64 B/line, 2-way
L2 Unified Cache	4 MB, 64 B/line, 1-way
Floating-Point Registers	32
Floating-Point Functional Units	2
Floating-Point Multiply Latency	4
Has Fused Multiply Add	No
Operating System	Tru64 v5.1B (rev.2650)
C Compiler	Compaq C v6.5-003
Fortran Compiler	GNU Fortran 3.3

TABLE 12(a)

DEC ALPHA 21264: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	72	4, 4, 72	4	0	1, 7, 1	1281
Model	84	4, 4, 84	4	0	0, 2, 2	1189
Unleashed	80					1491

TABLE 12(b)

DEC ALPHA 21264: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	148s	101s
Optimization Parameters	556s	
Total	704s	101s

TABLE 12(c)

DEC ALPHA 21264: TIMINGS

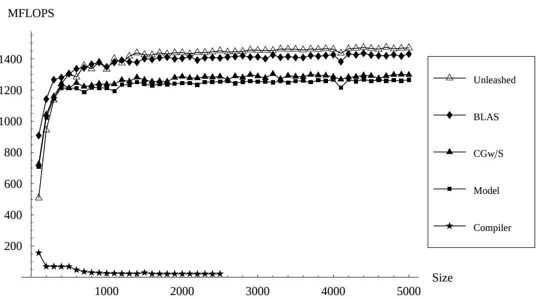

Fig. 12(d). DEC Alpha 21264: MMM Performance

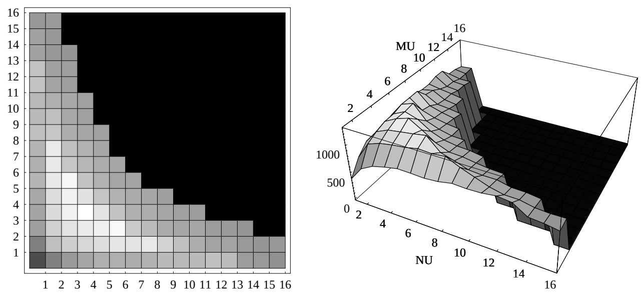

Fig. 12(e). DEC Alpha 21264: Sensitivity of performance to M_U and N_U

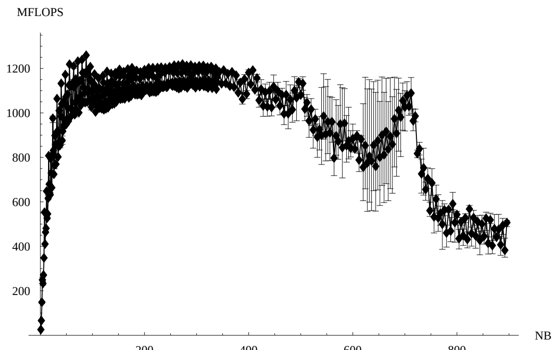

Fig. 12(f). DEC Alpha 21264: Sensitivity of performance to N_B

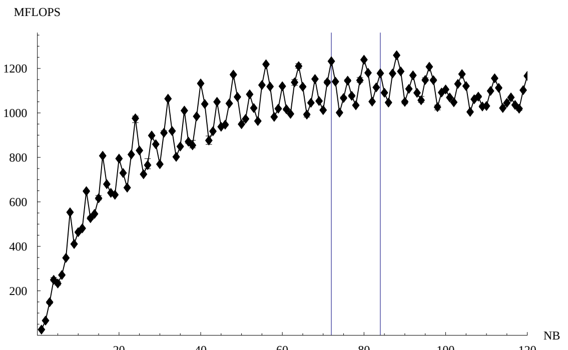

Fig. 12(g). DEC Alpha 21264: Sensitivity of performance to N_B (zoomed)

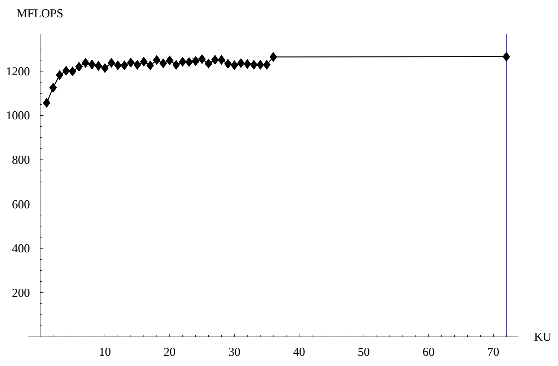

Fig. 12(h). DEC Alpha 21264: Sensitivity of performance to K_U

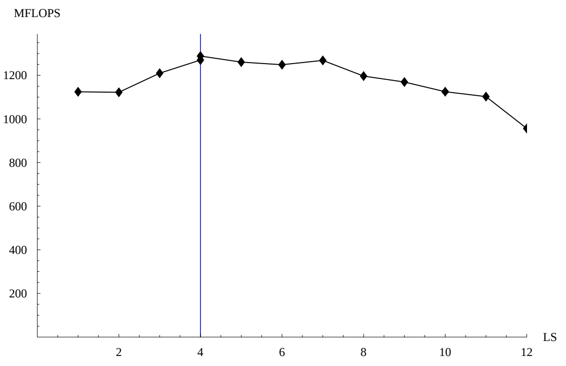

Fig. 12(i). DEC Alpha 21264: Sensitivity of performance to L_s

Feature	Value
Architecture	Out-Of-Order, RISC
CPU Core Frequency	375 MHz
L1 Data Cache	64 KB, 128 B/line, 128-way
L1 Instruction Cache	32 KB, 128 B/line, 128-way
L2 Unified Cache	4 MB, 128 B/line, ???-way
Floating-Point Registers	32
Floating-Point Functional Units	2
Floating-Point Multiply Latency	4
Has Fused Multiply Add	Yes
Operating System	AIX
C Compiler	XL C for AIX v.5
Fortran Compiler	XL Fortran for AIX

TABLE 13(a)

IBM POWER 3: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	80	4, 5, 80	6	1	0, 8, 1	1264
Model	84	4, 4, 84	4	1	0, 2, 2	1225
Unleashed	80					1257

TABLE 13(b)

IBM POWER 3: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	139s	154s
Optimization Parameters	1984s	
Total	2123s	154s

TABLE 13(c)

IBM POWER 3: TIMINGS

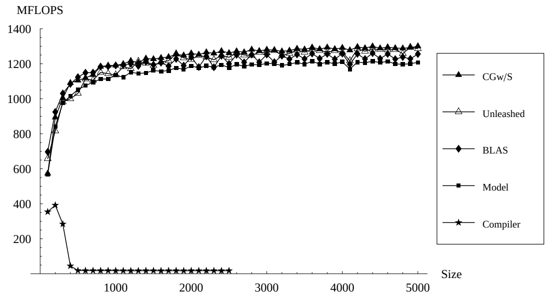

Fig. 13(d). IBM Power 3: MMM Performance

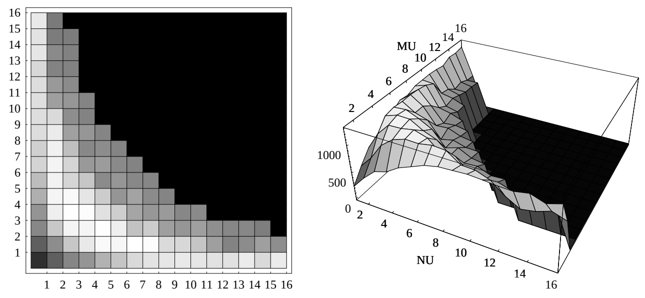

Fig. 13(e). IBM Power 3: Sensitivity of performance to M_U and N_U

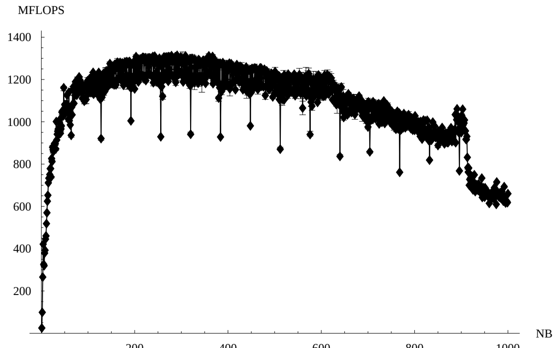

Fig. 13(f). IBM Power 3: Sensitivity of performance to N_B

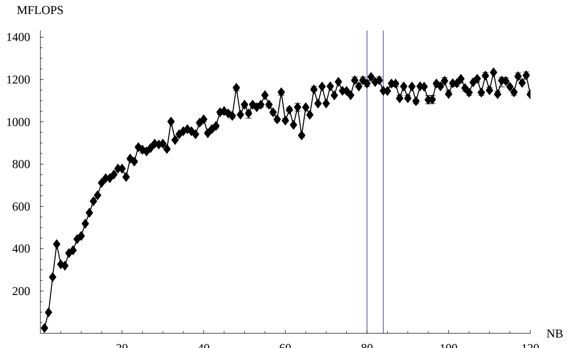

Fig. 13(g). IBM Power 3: Sensitivity of performance to N_B (zoomed)

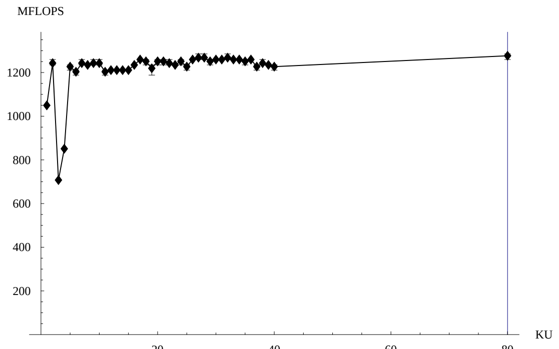

Fig. 13(h). IBM Power 3: Sensitivity of performance to K_U

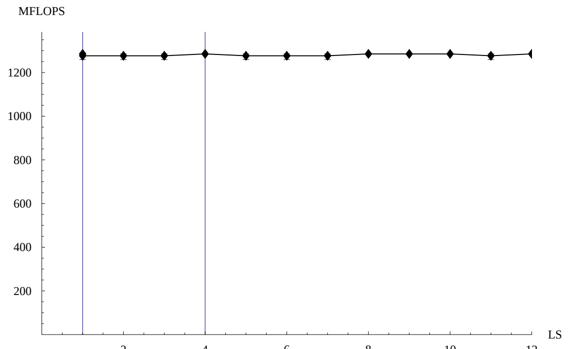

Fig. 13(i). IBM Power 3: Sensitivity of performance to L_s

Feature	Value
Architecture	Out-Of-Order, RISC
CPU Core Frequency	1450 MHz
L1 Data Cache	32 KB, 128 B/line, 2-way
L1 Instruction Cache	64 KB, 128 B/line, 1-way
L2 Unified Cache	1.5 MB, 128 B/line, 8-way
L3 Cache	32 MB, 512B/line, 8-way
Floating-Point Registers	32
Floating-Point Functional Units	2
Floating-Point Multiply Latency	4
Has Fused Multiply Add	Yes
Operating System	AIX
C Compiler	XL C for AIX v.5
Fortran Compiler	XL Fortran for AIX

TABLE 14(a)
IBM POWER 4: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	64	4, 4, 64	1	1	1, 8, 1	3468
Model	56	4, 4, 56	6	1	0, 2, 2	3400
Unleashed	64					3468

TABLE 14(b)
IBM POWER 4: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	175s	125s
Optimization Parameters	2390s	
Total	2665s	125s

TABLE 14(c)
IBM POWER 4: TIMINGS

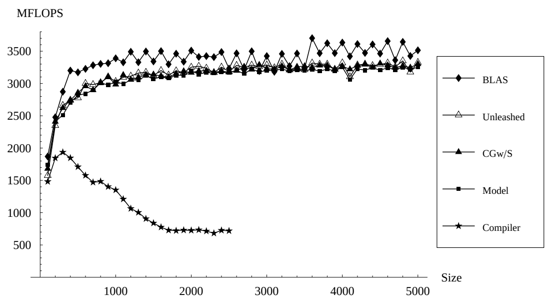

Fig. 14(d). IBM Power 4: MMM Performance

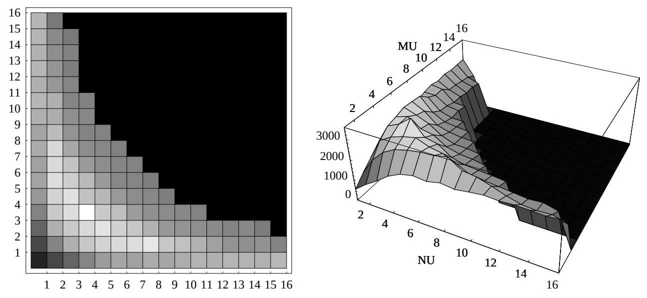

Fig. 14(e). IBM Power 4: Sensitivity of performance to M_U and N_U

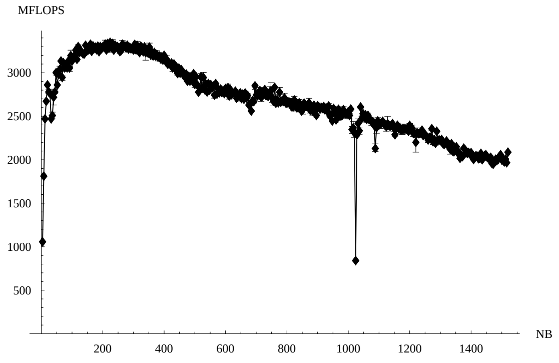

Fig. 14(f). IBM Power 4: Sensitivity of performance to N_B

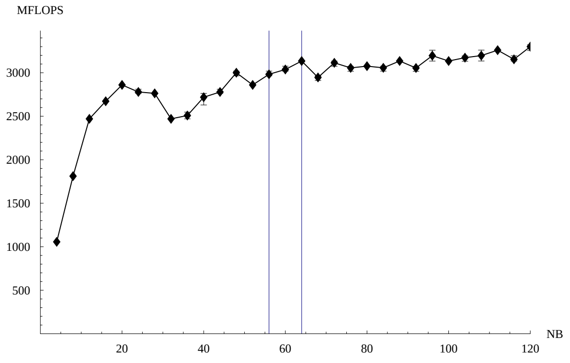

Fig. 14(g). IBM Power 4: Sensitivity of performance to N_B (zoomed)

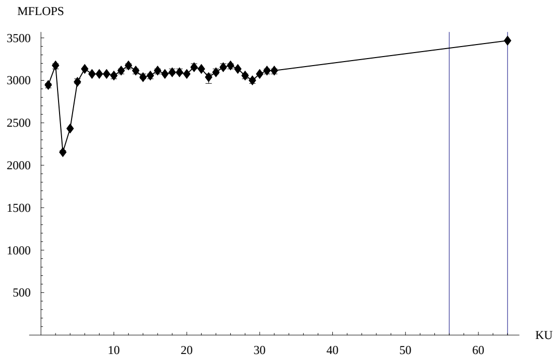

Fig. 14(h). IBM Power 4: Sensitivity of performance to K_U

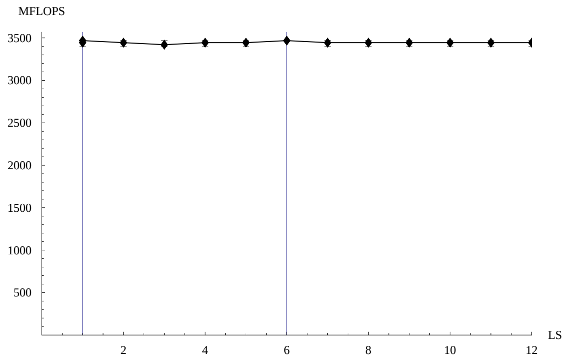

Fig. 14(i). IBM Power 4: Sensitivity of performance to L_s

Feature	Value
Architecture	Out-Of-Order, RISC
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	Yes
Operating System	IRIX64
C Compiler	SGI MIPSPro C 7.3.1.1m
Fortran Compiler	SGI MIPSPro FORTRAN 7.3.1.1m

TABLE 15(a)

SGI R12K: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	64	4, 5, 32	3	0	1, 8, 1	459
Model	58	5, 4, 58	1	1	0, 2, 2	442
Unleashed	64					464

TABLE 15(b)

SGI R12K: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	251s	117s
Optimization Parameters	5015s	
Total	5266s	117s

TABLE 15(c)

SGI R12K: TIMINGS

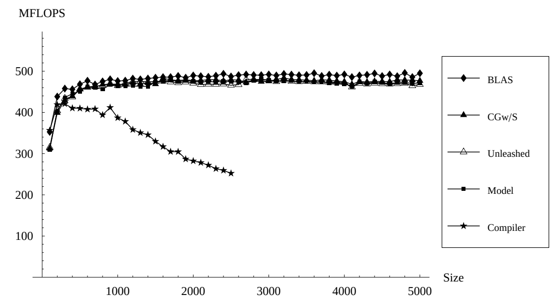

Fig. 15(d). SGI R12K: MMM Performance

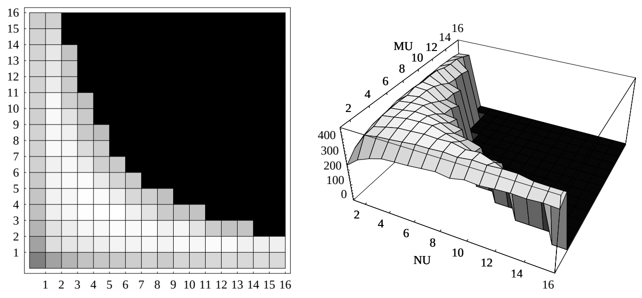

Fig. 15(e). SGI R12K: Sensitivity of performance to M_U and N_U

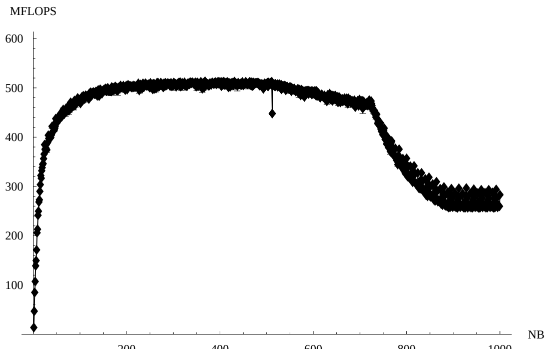

Fig. 15(f). SGI R12K: Sensitivity of performance to N_B

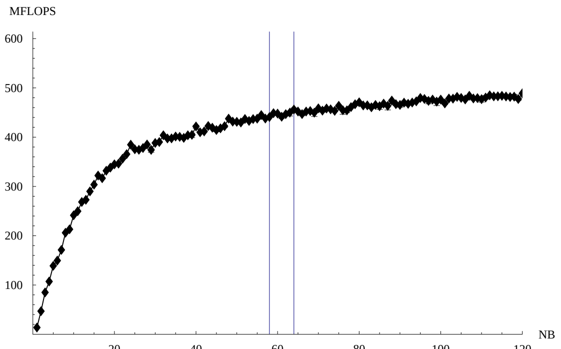

Fig. 15(g). SGI R12K: Sensitivity of performance to N_B (zoomed)

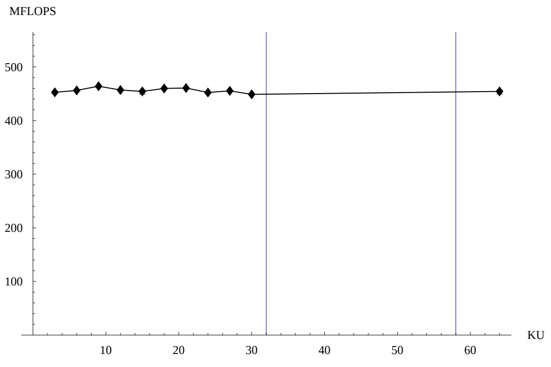

Fig. 15(h). SGI R12K: Sensitivity of performance to K_U

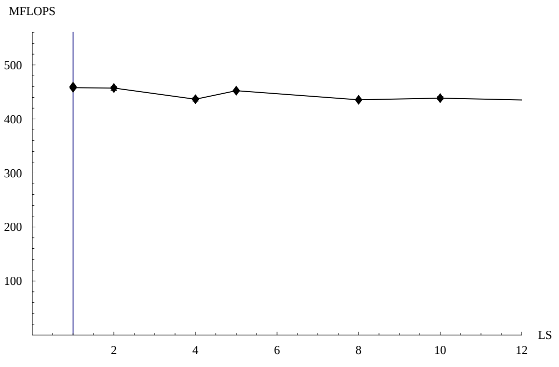

Fig. 15(i). SGI R12K: Sensitivity of performance to L_s

Feature	Value
Architecture	Out-Of-Order, RISC
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

TABLE 16(a)

SUN ULTRASPARC IIII: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	44	4, 3, 44	5	0	0, 3, 2	986
Model	84	4, 4, 84	4	0	0, 2, 2	1149
Unleashed	168					1695

TABLE 16(b)

SUN ULTRASPARC IIII: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	203s	112s
Optimization Parameters	1254s	
Total	1457s	112s

TABLE 16(c)

SUN ULTRASPARC IIII: TIMINGS

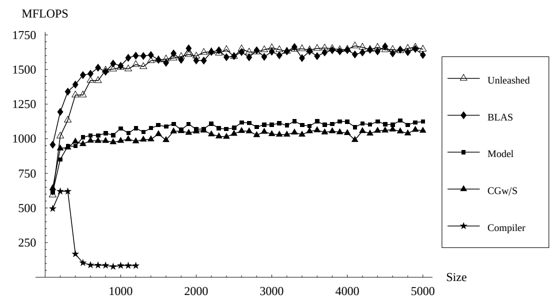

Fig. 16(d). Sun UltraSPARC IIIi: MMM Performance

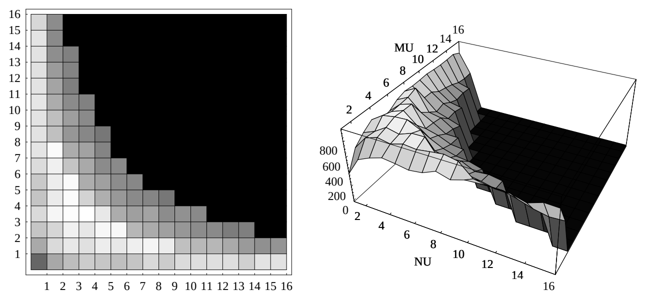

Fig. 16(e). Sun UltraSPARC IIIi: Sensitivity of performance to M_U and N_U

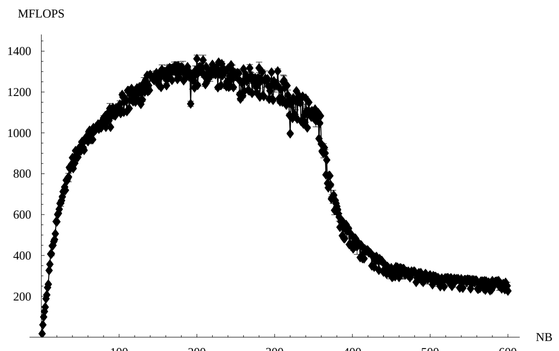

Fig. 16(f). Sun UltraSPARC IIIi: Sensitivity of performance to N_B

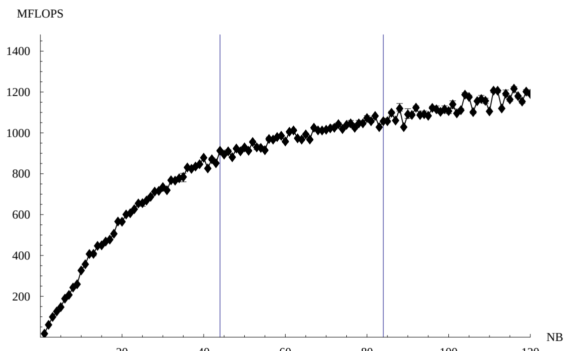

Fig. 16(g). Sun UltraSPARC IIIi: Sensitivity of performance to N_B (zoomed)

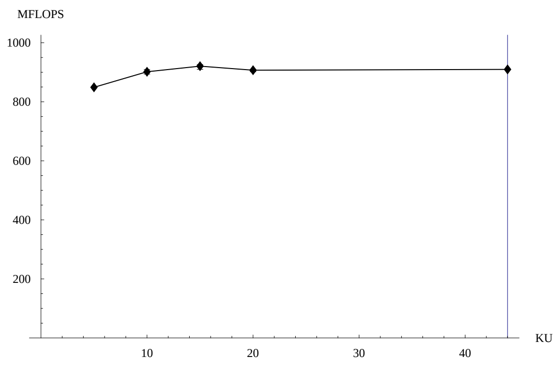

Fig. 16(h). Sun UltraSPARC IIIi: Sensitivity of performance to K_U

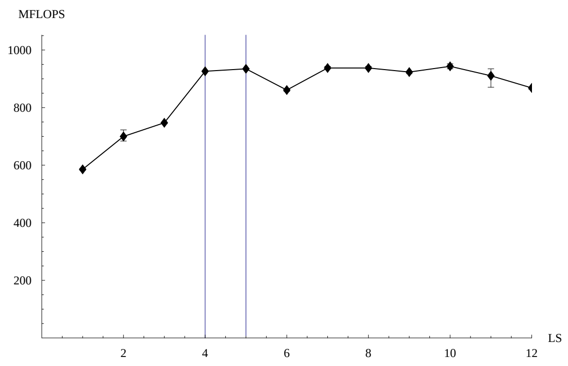

Fig. 16(i). Sun UltraSPARC IIIi: Sensitivity of performance to L_s

Feature	Value
Architecture	In-Order, EPIC, IA-64
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	3 MB, 128B/line, 12-way
Floating-Point Registers	128
Floating-Point Functional Units	2
Floating-Point Multiply Latency	4
Has Fused Multiply Add	Yes
Operating System	Linux 2.4.18-e.31smp
C Compiler	GNU C/C++ 3.3
Fortran Compiler	GNU Fortran 3.3

TABLE 17(a)

INTEL ITANIUM 2: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	80	10, 10, 4	4	1	0, 19, 1	4028
Model	30	10, 10, 8	1	1	0, 2, 2	1806
Unleashed	120					4891

TABLE 17(b)

INTEL ITANIUM 2: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	1555s	143s
Optimization Parameters	30710s	
Total	32265s	143s

TABLE 17(c)

INTEL ITANIUM 2: TIMINGS

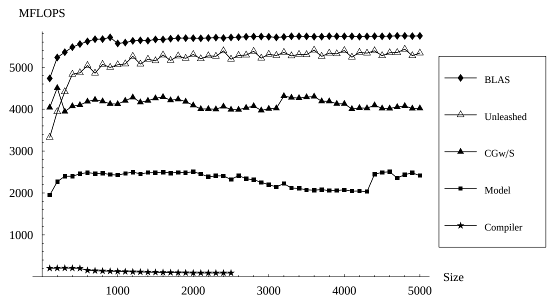

Fig. 17(d). Intel Itanium 2: MMM Performance

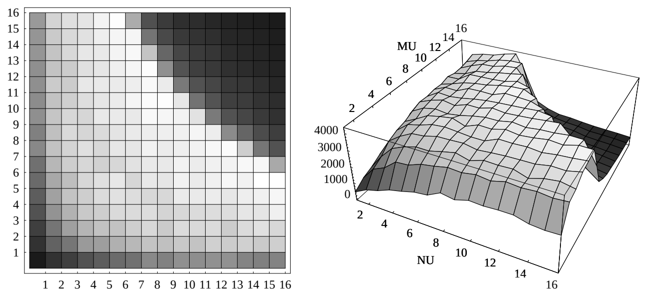

Fig. 17(e). Intel Itanium 2: Sensitivity of performance to M_U and N_U

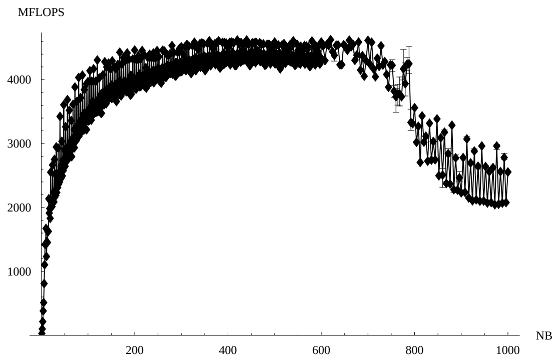

Fig. 17(f). Intel Itanium 2: Sensitivity of performance to N_B

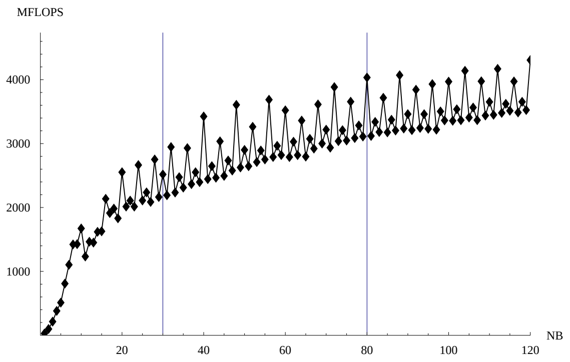

Fig. 17(g). Intel Itanium 2: Sensitivity of performance to N_B (zoomed)

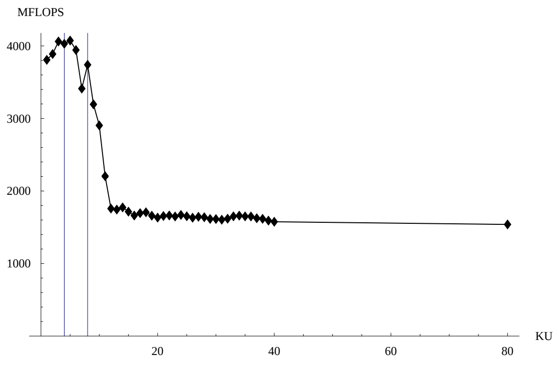

Fig. 17(h). Intel Itanium 2: Sensitivity of performance to K_U

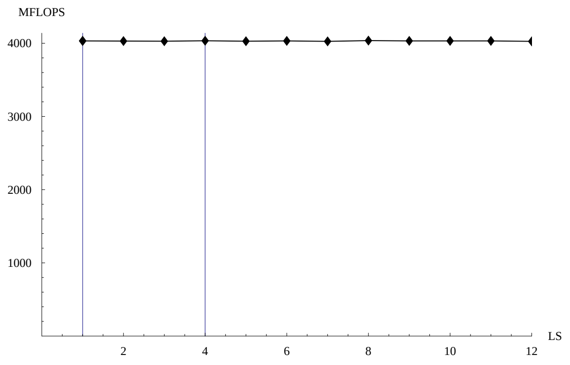

Fig. 17(i). Intel Itanium 2: Sensitivity of performance to L_s

Feature	Value
Architecture	Out-Of-Order, CISC, x86-64
CPU Core Frequency	1400 MHz
L1 Data Cache	64 KB, 64 B/line, 2-way
L1 Instruction Cache	64 KB, 64 B/line, 2-way
L2 Unified Cache	1024 MB, 64 B/line, 16-way
Floating-Point Registers	8 x87
Floating-Point Functional Units	ADD + MUL + Memory
Floating-Point Multiply Latency	4
Has Fused Multiply Add	No
Operating System	Linux 2.4.19
C Compiler	GCC C/C++ 3.3.2
Fortran Compiler	GNU Fortran 3.3.2

TABLE 18(a)

AMD OPTERON 240: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	60	6, 1, 60	6	1	0, 6, 1	2072
Model	88	2, 1, 88	2	0	0, 2, 2	1282
Unleashed	56					2608

TABLE 18(b)

AMD OPTERON 240: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	148s	101s
Optimization Parameters	556s	
Total	704s	101s

TABLE 18(c)

AMD OPTERON 240: TIMINGS

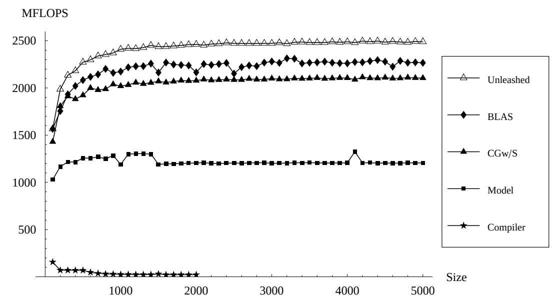

Fig. 18(d). AMD Opteron 240: MMM Performance

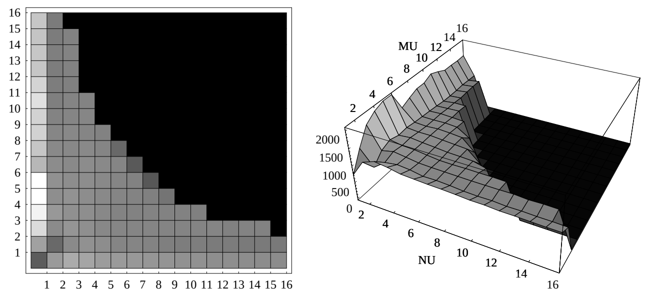

Fig. 18(e). AMD Opteron 240: Sensitivity of performance to M_U and N_U

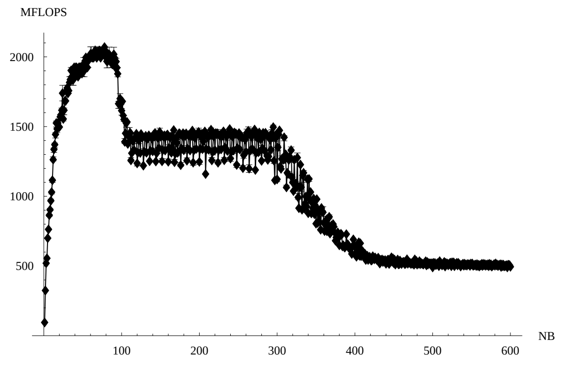

Fig. 18(f). AMD Opteron 240: Sensitivity of performance to N_B

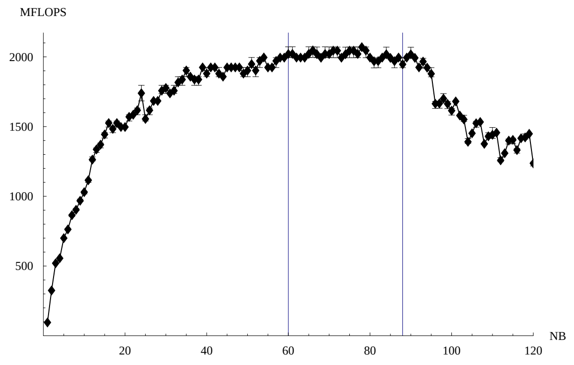

Fig. 18(g). AMD Opteron 240: Sensitivity of performance to N_B (zoomed)

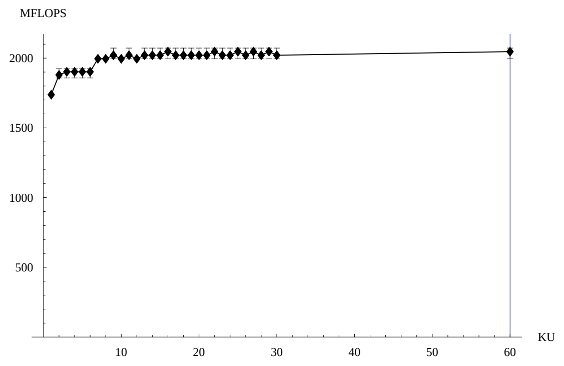

Fig. 18(h). AMD Opteron 240: Sensitivity of performance to K_U

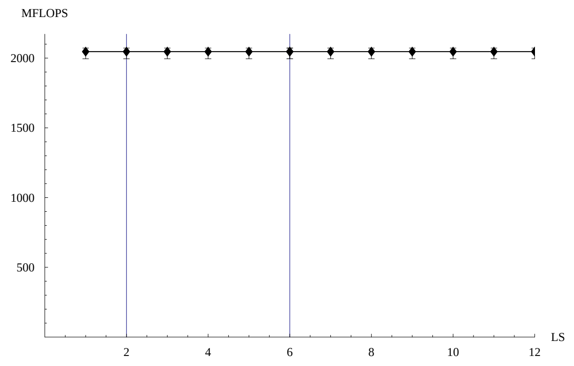

Fig. 18(i). AMD Opteron 240: Sensitivity of performance to L_s

Feature	Value
Architecture	Out-Of-Order, CISC, x86
CPU Core Frequency	1733 MHz
L1 Data Cache	64 KB, 64 B/line, 2-way
L1 Instruction Cache	64 KB, 64 B/line, 2-way
L2 Unified Cache	256 KB, 64 B/line, 16-way
Floating-Point Registers	8
Floating-Point Functional Units	ADD + MUL + Memory
Floating-Point Multiply Latency	4
Has Fused Multiply Add	No
Operating System	Linux 2.4.20
C Compiler	GNU C/C++ 3.2.2
Fortran Compiler	GNU Fortran 3.2.2

TABLE 19(a)

AMD ATHLON MP: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	76	4, 1, 76	1	0	0, 3, 2	1531
Model	88	2, 1, 88	2	0	0, 2, 2	1239
Unleashed	30					2512

TABLE 19(b)

AMD ATHLON MP: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	220s	121s
Optimization Parameters	3195s	
Total	3415s	121s

TABLE 19(c)

AMD ATHLON MP: TIMINGS

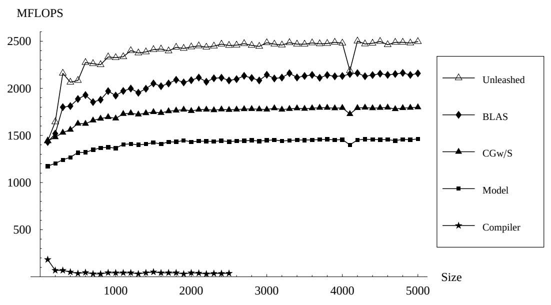

Fig. 19(d). AMD Athlon MP: MMM Performance

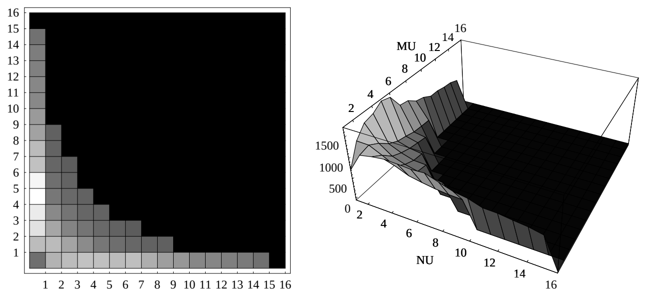

Fig. 19(e). AMD Athlon MP: Sensitivity of performance to M_U and N_U

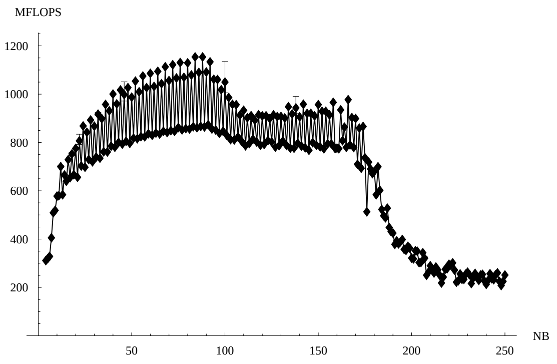

Fig. 19(f). AMD Athlon MP: Sensitivity of performance to N_B

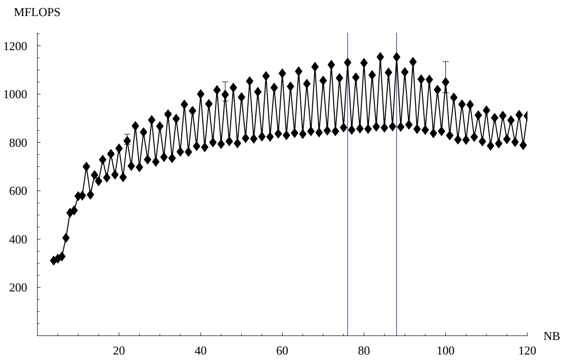

Fig. 19(g). AMD Athlon MP: Sensitivity of performance to N_B (zoomed)

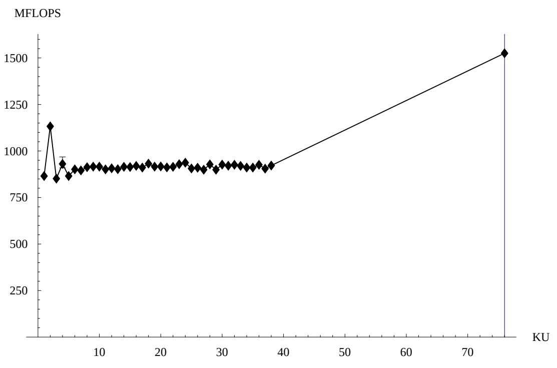

Fig. 19(h). AMD Athlon MP: Sensitivity of performance to K_U

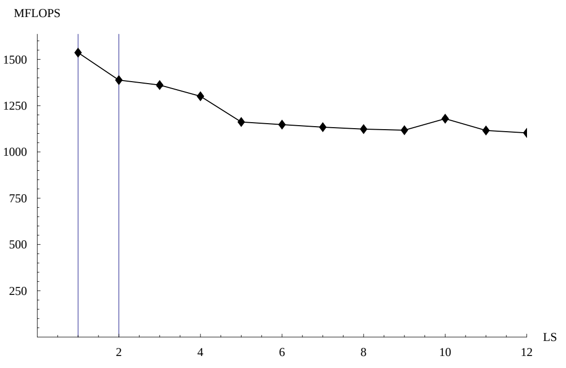

Fig. 19(i). AMD Athlon MP: Sensitivity of performance to L_s

Feature	Value
Architecture	Out-Of-Order, CISC, x86
CPU Core Frequency	1266 MHz
L1 Data Cache	16 KB, 32 B/line, 4-way
L1 Instruction Cache	16 KB, 32 B/line, 4-way
L2 Unified Cache	512 MB, 32 B/line, 8-way
Floating-Point Registers	8
Floating-Point Functional Units	1
Floating-Point Multiply Latency	5
Has Fused Multiply Add	No
Operating System	Linux 2.4.20-28.8smp
C Compiler	GNU C/C++ 3.2
Fortran Compiler	GNU Fortran 3.2

TABLE 20(a)

PENTIUM III: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	44	4, 1, 44	3	0	0, 3, 2	894
Model	42	2, 1, 42	2	0	0, 2, 2	841
Unleashed	40					951

TABLE 20(b)

PENTIUM III: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	133s	100s
Optimization Parameters	630s	
Total	763s	100s

TABLE 20(c)

PENTIUM III: TIMINGS

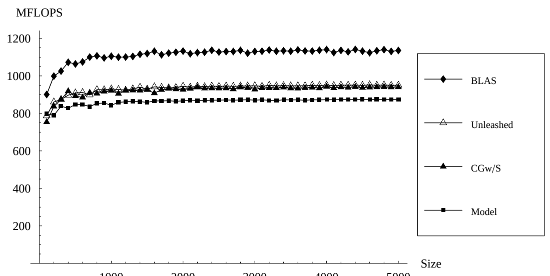

Fig. 20(d). Pentium III: MMM Performance

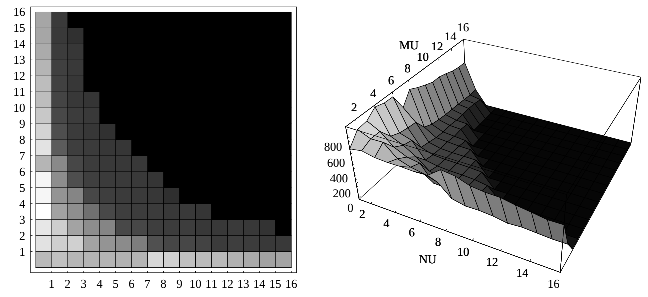

Fig. 20(e). Pentium III: Sensitivity of performance to M_U and N_U

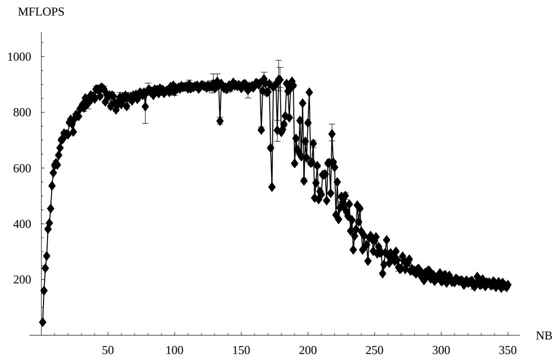

Fig. 20(f). Pentium III: Sensitivity of performance to N_B

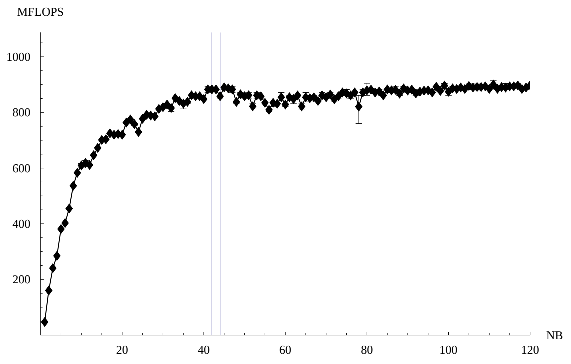

Fig. 20(g). Pentium III: Sensitivity of performance to N_B (zoomed)

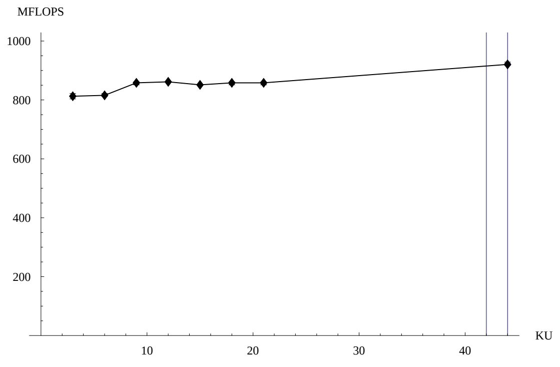

Fig. 20(h). Pentium III: Sensitivity of performance to K_U

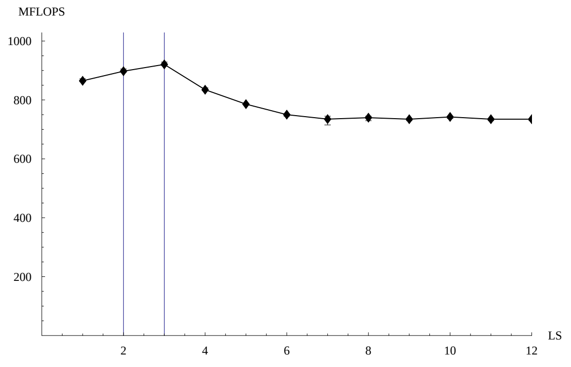

Fig. 20(i). Pentium III: Sensitivity of performance to L_s

Feature	Value
Architecture	Out-Of-Order, CISC, x86
CPU Core Frequency	2000 MHz
L1 Data Cache	8 KB, 64 B/line, 4-way
L1 Instruction Cache	12 K uOPs, 6 uOPs/line, 8-way
L2 Unified Cache	512 KB, 128 B/line, 8-way
Floating-Point Registers	8
Floating-Point Functional Units	1
Floating-Point Multiply Latency	7
Has Fused Multiply Add	No
Operating System	Linux 2.4.20-30.9smp
C Compiler	GNU C v3.2.2
Fortran Compiler	GNU Fortran 3.2.2

TABLE 21(a)
PENTIUM 4: PLATFORM SPECIFICATION

	N_B	M_U, N_U, K_U	L_s	FMA	F_F, I_F, N_F	MFLOPS
CGw/S	28	3, 1, 28	1	0	0, 2, 1	1504
Model	30	1, 1, 30	4	0	0, 2, 2	913
Unleashed	72					3317

TABLE 21(b)
PENTIUM 4: OPTIMIZATION PARAMETERS

	Search	Model
Machine Parameters	136s	98s
Optimization Parameters	643s	
Total	779s	98s

TABLE 21(c)
PENTIUM 4: TIMINGS

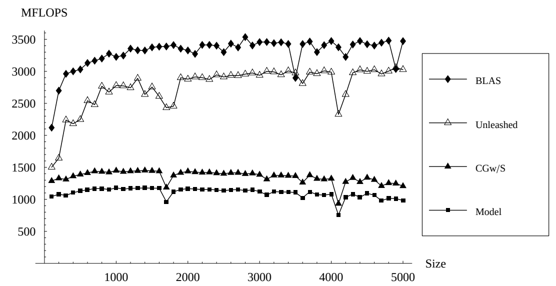

Fig. 21(d). Pentium 4: MMM Performance

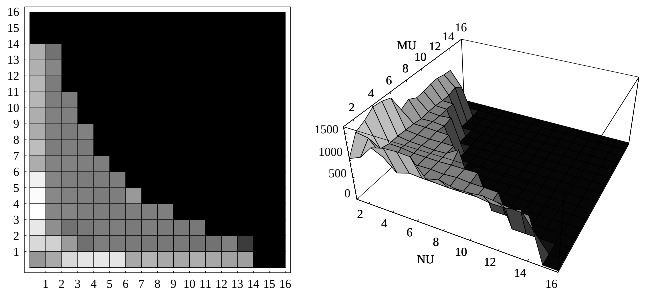

Fig. 21(e). Pentium 4: Sensitivity of performance to M_U and N_U

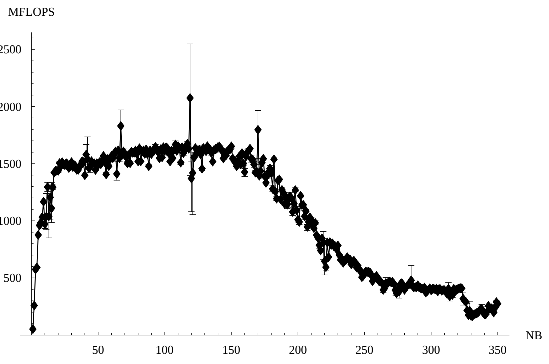

Fig. 21(f). Pentium 4: Sensitivity of performance to N_B

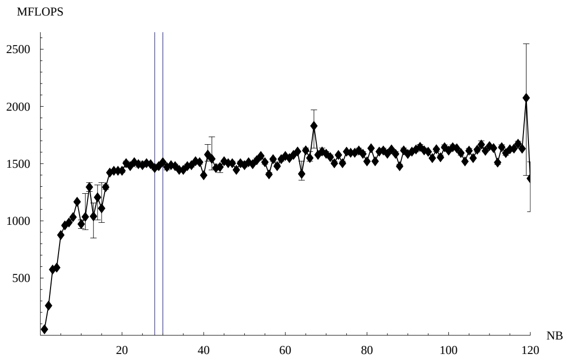

Fig. 21(g). Pentium 4: Sensitivity of performance to N_B (zoomed)

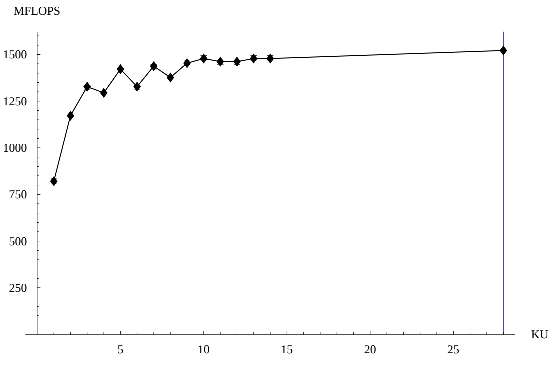

Fig. 21(h). Pentium 4: Sensitivity of performance to K_U

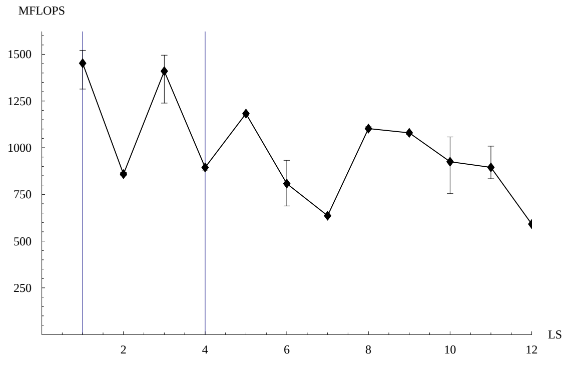

Fig. 21(i). Pentium 4: Sensitivity of performance to L_s