# An Experimental Comparison of Cache-oblivious and Cache-conscious Programs *

Kamen Yotov, Tom Roeder, Keshav Pingali
Cornell University
{kyotov,tmroeder,pingali}@cs.cornell.edu

John Gunnels, Fred Gustavson
IBM T. J. Watson Research Center
{gunnels,fg2}@us.ibm.com

## ABSTRACT

Cache-oblivious algorithms have been advanced as a way of circumventing some of the difficulties of optimizing applications to take advantage of the memory hierarchy of modern microprocessors. These algorithms are based on the divide-and-conquer paradigm – each division step creates sub-problems of smaller size, and when the working set of a sub-problem fits in some level of the memory hierarchy, the computations in that sub-problem can be executed without suffering capacity misses at that level. In this way, divide-and-conquer algorithms adapt automatically to all levels of the memory hierarchy; in fact, for problems like matrix multiplication, matrix transpose, and FFT, these recursive algorithms are optimal to within constant factors for some theoretical models of the memory hierarchy.

An important question is the following: how well do carefully tuned cache-oblivious programs perform compared to carefully tuned cache-conscious programs for the same problem? Is there a price for obliviousness, and if so, how much performance do we lose? Somewhat surprisingly, there are few studies in the literature that have addressed this question.

This paper reports the results of such a study in the domain of dense linear algebra. Our main finding is that in this domain, even highly optimized cache-oblivious programs perform significantly worse than corresponding cache-conscious programs. We provide insights into why this is so, and suggest research directions for making cache-oblivious algorithms more competitive.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: compilers, code generation, optimization; G.4 [**Mathematical Software**]

## General Terms

Algorithms, Measurement, Performance, Experimentation

## Keywords

Memory hierarchy, Memory Latency, Memory bandwidth, Cache-oblivious algorithms, Cache-conscious algorithms, Numerical Software

## 1. INTRODUCTION

The contributions of this paper are the following.

- We present detailed experiments on a number of high-performance platforms that show that even highly optimized recursive cache-oblivious programs may perform significantly worse than highly optimized cache-conscious programs for the same problem.
- We argue that part of the performance problem arises because the schedule of operations in recursive codes may be sub-optimal for exploiting processor pipelines. We show that the schedule of operations in iterative codes can make better use of processor pipelines.
- We argue that the rest of the performance problem arises from memory latency. Using analytical models, we point out that cache blocking serves two purposes: it can reduce the effective latency of memory requests and it can reduce the bandwidth required from memory. We argue quantitatively that I/O optimal algorithms [20, 25] may ameliorate the bandwidth problem, but their performance may still suffer from memory latency. In highly tuned iterative cache-conscious codes, the effective latency of memory requests is reduced by pre-fetching. We believe that this is needed in cache-oblivious programs as well.

### 1.1 Memory Hierarchy Problem

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

Both problems can be addressed by using caching – if most memory requests are satisfied by some cache level, the effective memory latency as well as the bandwidth required from memory are reduced. As is well known, the effectiveness of

*This work is supported in part by NSF grants 0615240, 0541193, 0509307, 0509324 and 0406380, as well as grants from the IBM and Intel Corportations


*SPAA'07,* June 9–11, 2007, San Diego, California, USA.

caching for a problem depends both on the algorithm used to solve the problem, and on the program used to express that algorithm (simply put, an algorithm defines only the dataflow of the computation, while a program for a given algorithm also specifies the schedule of operations and may perform storage allocation consistent with that schedule). One useful quantity in thinking about these issues is *algorithmic data reuse*, which is an abstract measure of the number of accesses made to a typical memory location by the algorithm. For example, the standard algorithm for multiplying matrices of size $n \times n$ performs $O(n^3)$ operations on $O(n^2)$ data, so it has excellent algorithmic data reuse since each data element is accessed $O(n)$ times; in contrast, matrix transpose performs $O(n^2)$ operations on $O(n^2)$ data, so it has poor algorithmic data reuse. When an algorithm has substantial algorithmic data reuse, the challenge is to write the program so that the memory accesses made by that program exhibit both spatial and temporal locality. In contrast, programs that encode algorithms with poor algorithmic data reuse must be written so that spatial locality is exploited.

## 1.2 Programming styles

Two programming styles are common in the domain of dense linear algebra: *iterative* and *recursive*.

In the iterative programming style, computations are implemented as nested loops. It is well known that naïve programs written in this style exhibit poor temporal locality and do not exploit caches effectively. Temporal locality can be improved by tiling the loops either manually or with restructuring compilers [22, 28]. The resulting program can be viewed as a computation over block matrices; tile sizes must be chosen so that the working set of each block computation fits in cache [12, 24]. If there are multiple cache levels, it may be necessary to tile for each one. Tiling for registers requires loop unrolling [2]. Since tile sizes are a function of cache capacity, and loop unroll factors depend on the number of available registers and on instruction cache capacity, this style of coding is called *cache-conscious* programming because the code either explicitly or implicitly embodies parameters whose optimal values depend on the architecture[1]. Simple architectural models or empirical search can be used to determine these optimal values [13, 27, 29]. Cache-conscious programs for dense linear algebra problems have been investigated extensively by the numerical linear algebra community and the restructuring compiler community. The Basic Linear Algebra Subroutine (BLAS) [1] libraries produced by most vendors are cache-conscious iterative programs, as are the matrix factorization routines in libraries like LAPACK [3].

In the recursive programming style, computations are implemented with divide-and-conquer. For example, to multiply two matrices A and B, we can divide the larger matrix (say A) into two sub-matrices $A_1$ and $A_2$, and multiply $A_1$ and $A_2$ by B; the base case of this recursion is reached when both A and B have a single element. Programs written in this divide-and-conquer style perform *approximate* blocking in the following sense. Each division step generates subproblems of smaller size, and when the working set of some sub-problem fits in a given level of cache, the computation can take place without suffering capacity misses at that level. The resulting blocking is only approximate since the size of the working set may be smaller than the capacity of the cache.

[1]Strictly speaking, these codes are both processor-conscious and cache-conscious, but we will use standard terminology and just call them cache-conscious.

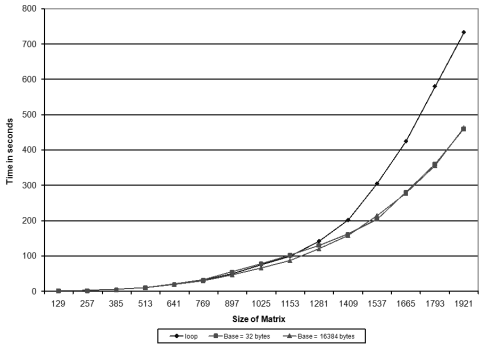



**Figure 1: An empirical study of recursive and iterative matrix multiplication programs**

An important theoretical result about divide-and-conquer algorithms was obtained in 1981 by Hong and Kung [20] who also introduced the *I/O model* to study the memory hierarchy performance of algorithms. This model considers a two level memory hierarchy consisting of a cache and main memory. The processor can compute only with values in the cache, so it is necessary to move data between cache and memory during program execution. The *I/O complexity* of a program is an asymptotic measure of the total volume of data movement when that program is executed. Hong and Kung showed that divide-and-conquer programs for matrix multiplication and FFT are optimal under this measure. In 1997, Gustavson implemented highly optimized recursive versions of common linear algebra routines [19]. In 1999, Frigo et al. extended the Hong and Kung result to matrix transposition; they also coined the adjective *cache-oblivious* to describe these programs because cache parameters are not reflected in the code [16].

A natural question is the following: *in domains like dense linear algebra in which it is important to exploit caches, how well do highly-optimized cache-oblivious programs perform compared to highly-optimized cache-conscious programs*? Is there a performance penalty, in other words, that cache-oblivious recursive programs pay for the ability to adapt automatically to the memory hierarchy? Somewhat surprisingly, we have not found any definitive studies of this question in the literature; the few comparative studies that we have found have compared the performance of unoptimized recursive and iterative programs. For example, Figure 1 shows the results of one study that found that a recursive implementation of matrix multiplication on an Itanium-2 outperforms an iterative implementation [21]. However, careful examination of this graph shows that the recursive implementation runs at about 30 Mflops; in comparison, the cache-conscious iterative native BLAS on this machine runs at almost 6 GFlops, as we discuss later in this paper.

## 1.3 Organization of this paper

In this paper, we describe the results of a study of the relative performance of highly-optimized recursive and iterative programs for (i) matrix multiplication on several modern architectures: IBM Power 5, Sun UltraSPARC IIIi, and Intel

Itanium 2, and (ii) matrix transpose on a single processor of the IBM Blue Gene[2]. The Power 5 is an out-of-order RISC processor, the UltraSPARC is an in-order RISC processor, and the Itanium is a long instruction word processor.

Key hardware parameters and details of the software on these machines are shown in Table 1. The L2 cache on the IBM Blue Gene processor is small because it is a prefetch buffer. Most of the programs we evaluate are generated by a domain-specific compiler we are building called BRILA (Block Recursive Implementation of Linear Algebra). The compiler takes recursive descriptions of linear algebra problems, and produces optimized iterative or recursive programs as output. It also implements key optimizations like scalar replacement [8], register allocation and operation scheduling at the level of the C program; these optimizations can be turned on or off as desired. Wherever appropriate, we compared the code produced by BRILA with code in vendor libraries like IBM's ESSL.

The rest of this paper is organized as follows.

In Section 2, we motivate approximate blocking by giving a quantitative analysis of how blocking can reduce the required bandwidth from memory. This analysis provides a novel way of thinking about the I/O optimality.

In Section 3, we discuss the performance of naïve iterative and recursive programs for matrix multiplication. These programs are *processor-oblivious* because they do not exploit registers and pipelines in the processor; they are also cache-oblivious. Therefore, neither program performs well on any architecture.

In Sections 4 and 5, we evaluate approaches for making the recursive and iterative programs processor-conscious. The goal is to enable these programs to exploit registers and processor pipelines. This is accomplished by unrolling loops and recursive calls, generating long basic blocks of instructions that are called from the main computations. These long basic blocks are called *microkernels* in this paper. Microkernels also serve to reduce loop and recursive call overhead. We discuss a number of algorithms for register allocation and scheduling of the microkernels, which we have implemented in the BRILA compiler. The main finding is that we were unable to produce a microkernel for the recursive code that performed well, even after considerable effort. In contrast, microkernels from the iterative code obtain near-peak performance. Therefore, in the rest of our studies of matrix multiplication, we only used microkernels obtained from the iterative code.

In Section 6, we add cache-consciousness to the processor-conscious code obtained in the previous section. We study the performance of programs obtained by wrapping recursive and cache-blocked iterative outer control structures around the iterative microkernels from the previous section. We also measure the performance obtained by using the native BLAS on these machines. The main finding in this section is that prefetching is important to obtain better performance. While prefetching is easy if the outer control structure is iterative, it is not clear how to accomplish this if the outer control structure is recursive.

Section 7 presents some findings about matrix transposition on a single processor of the IBM Blue Gene machine. Section 8 concludes with ideas for improving the performance of cache-oblivious algorithms.

[2]We did not have time to study matrix transpose on the other architectures, but we expect the results to be similar.

# 2. APPROXIMATE BLOCKING

In this section, we give a quantitative estimate of the impact of blocking on effective memory latency as well as on the bandwidth required from memory. This analysis provides a novel way of looking at approximate blocking in cache-oblivious programs. As a running example, we use Matrix-Matrix Multiply (MMM) on the Intel Itanium 2 architecture. The Itanium 2 can execute 2 FMAs (fused multiply-adds) per cycle, so to multiply two $N \times N$ matrices, this platform would ideally spend $\frac{N^3}{2}$ cycles. However, any naïve version of matrix multiplication will take much longer because the processor spends most of its time waiting for memory loads to complete.

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

## 2.1 Upper Bound on $N_B$

We derive an upper bound on $N_B$ by requiring the size of the working set of the block computation to be less than the capacity of the cache, $C$. The working set depends on the schedule of operations, but it is bounded above by the size of the sub-problem. Therefore, the following inequality is a conservative approximation.

$$3N_B^2 \leq C \tag{1}$$

Better approximations exist: in particular, it can be shown that it is sufficient to keep in the cache just one of the three blocks, a row or column of another block, and single element of the third block [29, 27], but Inequality (1) is sufficient for our purpose.

## 2.2 Effect of Blocking on Latency

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

$$l = \left(1 - \frac{3}{4N_B \times L_C}\right) \times l_C + \frac{3}{4N_B \times L_C} \times l_M \tag{2}$$

Equation 2 shows that the expected latency decreases with increasing $N_B$, so latency is minimized by choosing the largest $N_B$ for which the working set fits in the cache. In practice, the expected memory latency computed from Equation 2 is somewhat pessimistic because loads can be overlapped with each other or with actual computations, reducing the effective values of $l_C$ and $l_M$. These optimizations are extremely important in the generation of the microkernels, as we describe in Section 4. Furthermore, hardware and software prefetching can also be used to reduce effective latency, as discussed in Section 6.

| | Itanium 2 | Power 5 | UltraSPARC IIIi | IBM Blue Gene |
|---|---|---|---|---|
| Vendor CC | Intel C 9.0 | IBM XLC 7.0 | Sun C 5.5 | XLC 8.1 |
| GCC | gcc 3.4.3 | gcc 3.4.3 | gcc 3.2.2 | n/a |
| OS Version | Linux 2.6.9 | IBM AIX 5.3 | Sun Solaris 9 | BLRTS V1R3M1 |
| PAPI Version | 3.0.8.1 | 3.0.8.1 | 3.0.8.1 | n/a |
| BLAS Version | Intel MKL 8.0 | ESSL 4.2.0.2 | Sun Studio 8 | ESSL 4.2.5 |
| processor Frequency | 1.5 GHz | 1.65 GHz | 1.06 GHz | 700 MHz |
| processor Peak Rate | 6.0 GFlops | 6.6 GFlops | 2.12 GFlops | 2.8 GFlops |
| Has FMA | Yes | Yes | No | Yes |
| Has RegRelAddr | No | Yes | Yes | Yes |
| # of Registers | 128 | 32 | 32 | 32 16-byte SIMD |
| L1 Size | 16 kB | 32 kB | 64 kB | 32 kB |
| L1 Line Size | 64 B | 128 B | 32 B | 32 B |
| L2 Size | 256 kB | 1.875 MB | 1 MB | 2 kB |
| L2 Line Size | 128 B | 128 B | 32 B | 128 B |
| L3 Size | 3 MB | 36 MB | n/a | 4 MB |
| L3 Line Size | 128 B | 512 B | n/a | 128 B |
| I-Cache Size | 16 kB | 64 kB | 32 kB | 32 kB |

**Table 1: Software and Hardware parameters**

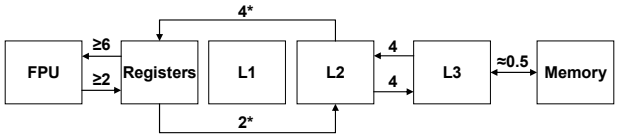



**Figure 2: Bandwidth of the Itanium 2 memory hierarchy, measured in doubles/cycle[3].**

## 2.3 Effect of Blocking on Bandwidth

In the restructuring compiler community, blocking is seen as a technique for reducing the effective latency of memory accesses. To understand the virtues of the cache-oblivious approach, it is better to view blocking as a technique for reducing the bandwidth required from memory.

Each FMA operation in MMM reads three values and writes one value. The required bandwidth to perform these reads and writes is $4N^3 \div \frac{N^3}{2} = 8$ doubles/cycle. Figure 2[3] shows the bandwidth between different levels of the memory hierarchy of the Itanium (floating-point values are not cached in the L1 cache on the Itanium). It can be seen that the register-file can sustain the required bandwidth but memory cannot.

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

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

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

[3]Floating-point values are not cached at L1 in Itanium 2, they are transferred directly to / from L2 cache. The L2 cache can transfer 4 values to floating point registers and 2 values from floating point registers per cycle, but there is a maximum total of 4 memory operations.

Inequalities 1 and 3 imply the following inequality for $N_B$:

$$\frac{8}{B(L1, M)} \leq N_B \leq \sqrt{\frac{C}{3}} \tag{4}$$

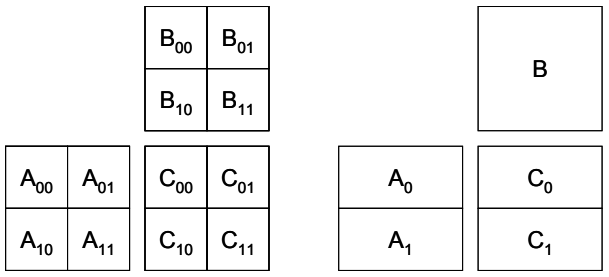
This argument generalizes to a multi-level memory hierarchy. If $B(L_i, L_{i+1})$ is the bandwidth between levels $i$ and $i+1$ in the memory hierarchy, $N_B(i)$ is the block size for the $i^{th}$ cache level, and $C_i$ is the capacity of this cache, we obtain the following inequality:

$$\frac{8}{B(L_i, L_{i+1})} \leq N_B(i) \leq \sqrt{\frac{C_i}{3}} \tag{5}$$

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

For the Itanium 2, we have seen that register blocking is needed to prevent the bandwidth between registers and L2 cache from becoming the bottleneck. If $N_B(R)$ is the size of the register block, we see that $\frac{8}{4} \leq N_B(R) \leq \sqrt{\frac{126}{3}}$. Therefore, $N_B(R)$ values between 2 and 6 will suffice. If we use $N_B(R)$ in this range, the bandwidth required from L2 to registers is between 1.33 and 4 doubles per cycle. Note that this much bandwidth is also available between the L2 and L3 caches. Therefore, it is not necessary to block for the L2 cache to ameliorate bandwidth problems. Unfortunately, this bandwidth exceeds the bandwidth between L3 cache and memory. Therefore, we need to block for the L3 cache. The appropriate inequality is $\frac{8}{0.5} \leq N_B(L_3) \leq \sqrt{\frac{4\text{MB}}{3}}$. Therefore, $N_B(L_3)$ values between 16 and 418 will suffice.

Thus, for the Itanium 2, there is a range of block sizes that can be used. Since the upper bound in each range is more than twice the lower bound, the approximate blocking of a divide-and-conquer implementation of a cache-oblivious program will generate sub-problems in these ranges, and




$C_{00} \leftarrow A_{00} \times B_{00} + A_{01} \times B_{10}$
$C_{01} \leftarrow A_{01} \times B_{11} + A_{11} \times B_{01}$
$C_{11} \leftarrow A_{11} \times B_{01} + A_{10} \times B_{01}$
$C_{10} \leftarrow A_{10} \times B_{00} + A_{11} \times B_{10}$

(a) All Dimensions

$C_0 \leftarrow A_0 \times B$
$C_1 \leftarrow A_1 \times B$

(b) Largest Dimension

**Figure 3: Divide-and-conquer strategies for MMM**

therefore bandwidth is not a constraint. Of course, latency of memory accesses may still be a problem. In particular, since blocking by cache-oblivious programs is only approximate, the analysis of Section 2.2 suggests that reducing the impact of memory latency is more critical for cache-oblivious codes than it is for cache-conscious codes. We will revisit this point in more detail in Section 6.

# 3. NAÏVE CODES

In this section, we discuss naïve recursive and iterative programs that are oblivious to both the processor and the memory hierarchy. There are two high-level design decisions to be made when writing either program: what control structure and what data structure to use.

Figure 3 shows two recursive control structures for implementing matrix multiplication. A well-known approach is to bisect both A and B along rows and columns, generating eight independent sub-problems as shown in Figure 3(a). The recursion terminates when the matrices consist of single elements. For obvious reasons, we refer to this strategy as the *all-dimensions* (AD) strategy.

Bilardi et al. [7] have pointed out that it is possible to optimize memory hierarchy performance by using a Gray code order to schedule the eight sub-problems so that there is always one sub-matrix in common between successive sub-problems. One such order can be found in Figure 3(a) if the sub-problems are executed in left-to-right, top-to-bottom order. For example, the first two sub-problems have $C_{00}$ in common, and the second and third have $A_{01}$ in common.

An alternative control strategy is the *largest-dimension* (LD) strategy proposed by Frigo et al. [16], in which only the largest of the three dimensions is divided, as shown in Figure 3(b). Both the AD and LD strategies lead to programs that are optimal in the Hong and Kung I/O complexity model [20].

As a baseline for performance comparisons, we used the simple iterative version of matrix multiplication. The three loops in this program are fully permutable, so all six orders of the loop nest compute the same values. In our experiments, we used the *jki* order. For the experiments in this section, we chose row-major array order. Note that the *jki* loop order is the worst loop order for exploiting spatial locality if arrays stored in row-major order (as discussed in [14]). We chose this order to eliminate any advantage the iterative code might obtain from exploiting spatial locality.

As an aside, we mention that we investigated Morton-Z storage order [10] as an alternative to row-major order. Accessing array elements is substantially more complex for Morton-Z order, especially for matrices whose dimensions are not a power of two. Even for matrices whose dimensions are a power of two, we rarely found any improvement in performance. This finding is consistent with previous studies that have concluded that space-filling storage orders like Morton-Z order pay off only when the computation is out of core [6].

Figure 4 shows the results of performing complete MMMs on the machines in our study (for lack of space, we have consolidated all the performance graphs for each machine into a single graph). Since we explore a large number of implementations in this paper, we use a tuple to distinguish them, the first part of which describes the outer control structure.

- R – using the Recursive AD control structure;
- I – using a triply-nested Iterative control structure;

The second part of the tuple describes the microkernel, and it will be explained as the microkernels are developed in Sections 4 and 5. However, when the outer control structure invokes a single statement to perform the computations, we use the symbol S (for Statement). For completeness, we include performance lines for MMM performed by the Vendor BLAS using standard row-major storage format for the arrays.

With this notation, note that the lines labelled R S in Figure 4 shows the performance of the AD cache-oblivious program, while the lines labelled I S shows the performance of the nested loop program. Both programs perform very poorly, obtaining roughly 1% of peak on all the machines. As a point of comparison, vendor BLAS on the Itanium 2 achieves close to peak performance. The performance of LD was close to that of AD on all machines, so we do not discuss it further.

## 3.1 Discussion

To get some insight into why these programs perform so poorly, we studied the assembly listings and the values of various hardware counters on the four machines. This study revealed three important reasons for the poor performance.

- As is well-known, the major problem with the recursive program is the overhead of recursion, since each division step in the divide-and-conquer process involves a procedure call. Our measurements on the Itanium showed that this overhead is roughly 100 cycles per FMA, while on the UltraSPARC, it is roughly 360 cycles. This integer overhead is much less for the iterative program.
- A second reason for poor performance is that the programs make poor use of registers. Compilers do not track register values across procedure calls, so register blocking for the recursive code is difficult. In principle, compilers can perform register blocking for the iterative program, but none of the compilers were able to accomplish this.
- Finally, a remarkable fact emerges when we look at the number of L2 cache misses on the Itanium. Figure 5

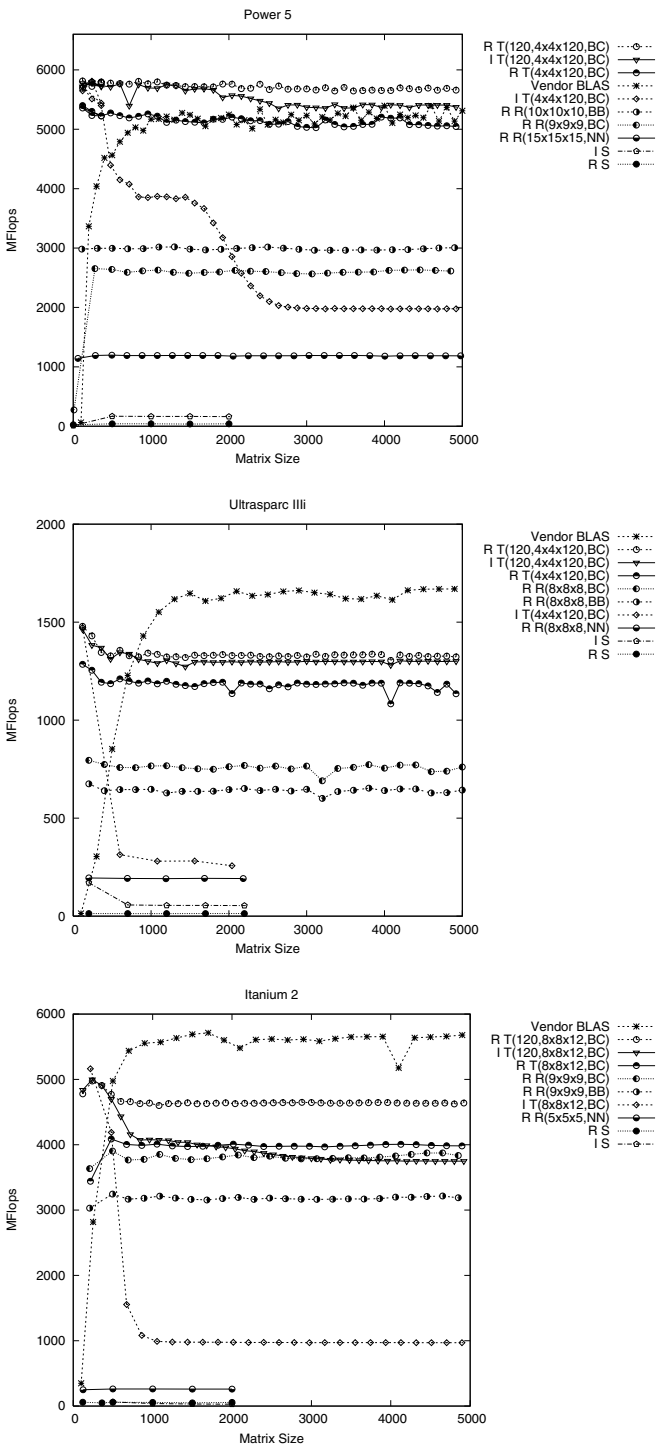



**Figure 4: Complete MMM performance**

shows the number of cache misses per FMA for the iterative and recursive programs.

The iterative program suffers roughly two misses per FMA. This makes intuitive sense because for the *jki* loop order, the accesses to $\mathsf{A}_{ik}$ and $\mathsf{C}_{ij}$ miss in the cache since the A and C arrays are stored in row-major order but are accessed in column-major order. The element

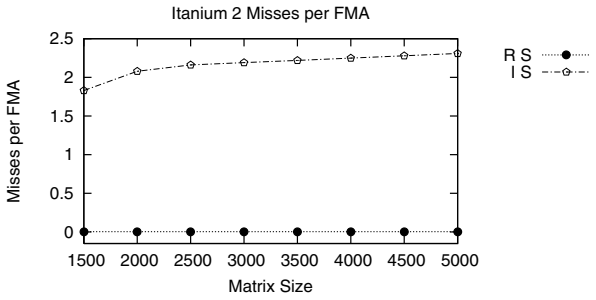



**Figure 5: Data cache misses per FMA instruction in MMM**

$\mathsf{B}_{kj}$ is invariant in the innermost loop, so it does not cause cache misses. Therefore, each iteration of the innermost loop performs one FMA, makes four data accesses, and misses on two of these accesses, resulting in a miss ratio of 0.5 and misses/FMA of 2. In short, poor memory hierarchy behavior limits the performance of the iterative code. Remarkably, the recursive program suffers only 0.002 misses per FMA, resulting in a miss ratio of 0.0005! This low miss ratio is a practical manifestation of the theoretical I/O optimality of the recursive program. Nevertheless, the poor performance of this code shows that I/O optimality alone does not guarantee good overall performance.

To improve performance, it is necessary to massage the recursive and iterative codes so that they become more processor-conscious and exploit the processor pipeline and the register file. Section 4 describes how processor-consciousness can be added to recursive codes. Section 5 describes how this can be done for iterative codes.

# 4. PROCESSOR-CONSCIOUS RECURSIVE CODES

To make the recursive program processor-conscious, we generate a long basic block of operations called a microkernel that is obtained by unrolling the recursive code completely for a problem of size $R_U \times R_U \times R_U$ [16]. The overall recursive program invoke this microkernel as its base case. There are two advantages to this approach. First, it is possible to perform register allocation and scheduling of operations in the microkernel, thereby exploiting registers and the processor pipeline. Second, the overhead of recursion is reduced.

We call the long basic block a *recursive* microkernel since the multiply-add operations are performed in the same order as they were in the original recursive code (there is no recursion in the code of the microkernel, of course). The optimal value of $R_U$ is determined empirically for values between 1 and 15.

Together with the control structure, one needs to worry about which data structure to use to represent the matrices. Virtually all high-performance BLAS libraries internally use a form of a blocked matrix, such as Row-Block-Row (RBR) [19]. An alternative is to use a recursive data layout, such as a space filling curve like Morton-Z [10]. We compared the MMM performance using both these choices and we rarely saw any performance improvement using Morton-Z order over RBR. Thus we use RBR in all experiments in this paper, and we chose the data block size to match our

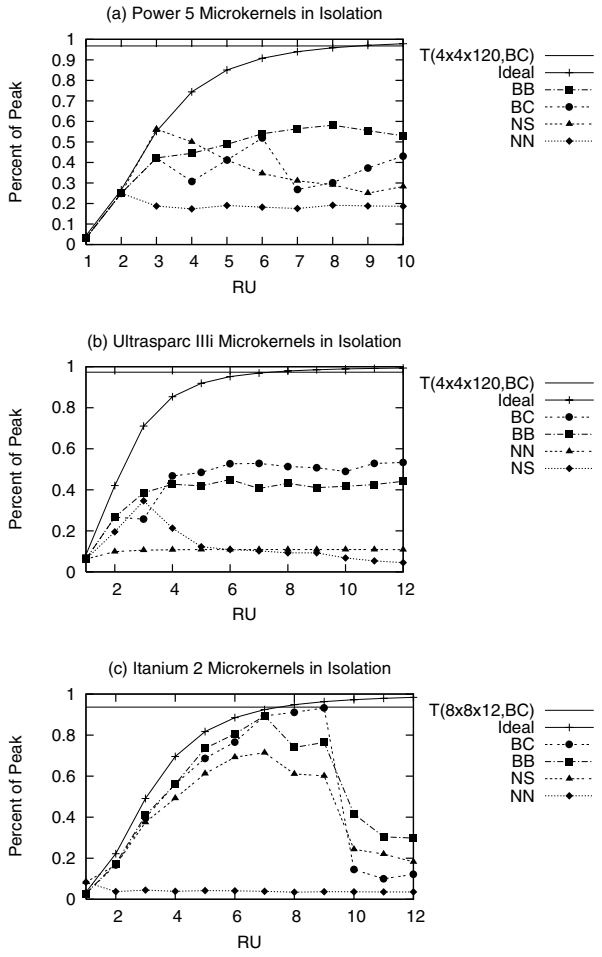



**Figure 6: Microkernel performance in isolation**

block size. Note however, that the native BLAS on all the machines use standard row-major order for the input arrays and copy these arrays internally into RBR format, so care should be taken in performance comparisons with the native BLAS.

We considered three different approaches to performing register allocation and scheduling for the microkernel.

## 4.1 R($R_U \times R_U \times R_U$, NN)

The first approach is to use the native compiler on each platform to compile the microkernel. We call this version R($R_U \times R_U \times R_U$, NN) because it is generated from Recursive inlining when data is $R_U$ along the three dimensions; NN stands for Native-None, and it means that the native compiler is used to schedule the code and no other register allocation is performed.

Figure 6 shows the performance of different microkernels in isolation on the four architectures of interest. Intuitively, this is the performance obtained by a microkernel if all of its memory accesses are satisfied by the highest cache level (L2 on the Itanium and L1 on the other machines). This performance is measured by invoking the microkernel repeatedly on the same data (the RBR format ensures that there are no conflict misses).

We focus on the UltraSPARC results. Figure 6(b) shows that the performance of the microkernel on the UltraSPARC in isolation is only about 11% of peak. This is also the performance of the complete MMM computation using this microkernel (190 MFlops out of 2.12 GFlops). The line labelled "ideal" corresponds to the highest performance one can achieve for a microkernel of given size, given the cost of ramping up and draining the computations of the microkernel. The line labelled "T(4x4x120,BC)" shows the performance of the best *iterative* microkernel, discussed in detail in Section 5.

Figure 4 shows that the overall performance is better than that of the naïve recursive version discussed in Section 3 because the overhead of recursion is amortized over the computations in the microkernel. An examination of the assembly code, however, showed that the compilers were not able to register allocate array elements. This optimization requires the compiler to discover whether or not the matrices are aliased. Even in the best production compilers, this alias analysis is often insufficient.

Some compilers can be told to assume that there is no aliasing in the microkernel. We found that the Intel C compiler (version 9.0) on the Itanium 2 was able to produce code comparable in performance to that of our most advanced recursive microkernel (Section 4.3) if it is told that there is no aliasing in the code for the microkernel.

## 4.2 R($R_U \times R_U \times R_U$, BB)

At Frigo's suggestion [15], we addressed this problem by implementing modules in the BRILA compiler that (i) used Belady's algorithm [4] to perform register allocation on the unrolled microkernel code, and then (ii) performed scheduling on the resulting code. Our implementation of Belady's algorithm is along the lines of [18]. This code was then compiled using the native compiler on each platform. In these experiments, we ensured that the native compiler was used only as a portable assembler, and that it did not perform any optimizations that interfered with BRILA optimizations.

The key idea behind using Belady's algorithm is that when it is necessary to spill a register, the value that will be needed furthest in the future should be evicted. This value is easy to discover in the context of microkernels, since we have one large basic block. The Belady register allocation algorithm is guaranteed to produce an allocation that results in the minimum number of loads. Different architectures require slightly different versions of the allocator. For instance, on the Itanium 2, Belady register allocation is implemented in two passes – one to allocate floating-point registers and a subsequent one to allocate integer registers. This division is necessary because the Itanium 2 architecture does not have a register-relative addressing mode, so the address of each memory operation needs to be pre-computed into an integer register. To decide on an allocation for the integer registers, we need to know the order of floating-point memory operations, but this order is not known before the floating-point registers themselves are allocated.

The BRILA scheduler is a simplified version of a general instruction scheduler found in compilers, since it has to handle only a basic block of floating-point FMAs (or multiplies and adds when the architecture does not have an FMA instruction), floating-point loads and stores, and potentially integer adds (for pointer arithmetic on Itanium 2). It accepts a simple description of the architecture and uses it to

- The scheduler works on blocks of the following instruction types:
  - Floating-point FMA, multiply, and add;
  - Floating-point load and store;
  - Integer arithmetic for address computation.
- The scheduler is parameterized by a description of the target architecture, which consists of:
  - HasFMA : bool – specifies whether the architecture has a floating-point FMA instruction.
  - HasRegRelAddr : bool – specifies whether the architecture supports register relative addressing mode, or all addresses need to computed into an integer register in advance (e.g. Itanium 2).
  - Latency : instruction → bool – specifies the latency in cycles of all instructions of interest.
  - Set of possible instruction bundles, each of which can be dispatched in a single cycle. The way we describe this set is by first mapping each instruction of interest to an instruction *type*. Each instruction type can be dispatched to one or more different execution *ports* inside the processor. Finally, the processor can dispatch at most one instruction to each execution port, for a subset of execution ports per cycle. We enumerate the possible sets of execution ports that can be dispatched together.
- The scheduler produces instruction bundles at each step as follows:
  1. Considers all instructions which have not been scheduled yet;
  2. Without changing the relative order of the instructions, removes all instructions from the list which depend on instructions that have not been scheduled yet;
  3. Greedily selects the largest subset of instructions from the resulting list which matches one of the subsets of execution ports the processor supports. It ensures that:
     - Instructions of the same type execute in program order;
     - Instructions from different types are given different execute preferences. The scheduler prefers to dispatch computational instructions most, followed by loads and stores, followed by integer arithmetic (if necessary).

**Figure 7: The BRILA instruction scheduler**

schedule the instructions appropriately. A brief description of the scheduler is presented in Figure 7.

We call the resulting microkernel, generated by using the Belady register allocation algorithm and the BRILA scheduler, R($R_U \times R_U \times R_U$, BB), where BB stands for BRILA-Belady.

Figure 6(b) shows that on the UltraSPARC, the performance in isolation of this microkernel is above 40% of peak for $R_U > 3$. The performance of the complete MMM is only at about 640 MFlops, or just about 32% of the 2 GFlops peak rate. Note that on the Itanium 2, register spills hurt the performance of this microkernel for $R_U > 7$. An even greater drop occurs for $R_U > 9$ because the microkernel overflows the I-Cache.

Interestingly, for the R($R_U \times R_U \times R_U$, NS) microkernel, Figure 4(a) shows that the IBM XLC Compiler at its highest optimization level is able to produce code which is slightly faster than the corresponding BB microkernel.

1. Generate the sequence of FMA operations in the same way we do this for R($R_U \times R_U \times R_U$, NN)
2. Generate an approximate schedule for this sequence:
   1. Consider the issue width of the processor for floating-point operations and schedule that many FMA instructions per cycle. Assume that an arbitrary number of other instructions (memory, integer arithmetic) can be executed in each cycle.
   2. If the processor has no FMA instruction, break each FMA into its two components, replace the FMA with its multiply part, and schedule its add part for Latency (multiply) cycles later;
   3. Assume an infinite virtual register file, and allocate each operand of each computational floating-point instruction (FMA, multiply, or add) into a different virtual register.
      1. Schedule a memory load into a register Latency (load) cycles before the FMA (or multiply) using the corresponding value.
      2. Schedule a memory store from a register Latency (FMA) (or Latency (add)) cycles after the FMA (or add) that modifies that register.
      3. Whenever the life spans of two registers that hold the same physical matrix element overlap, we merge them into a single virtual register and eliminate unnecessary intermediate loads and stores. This step is required to preserve the semantics of the microkernel.
      4. Additionally, when the life spans of two registers that hold the same physical matrix element do not overlap, but are close (say at most $\delta$ cycles apart), we still merge them to take advantage of this reuse opportunity. This step is not required to preserve the correctness of the program, but can allow significant reuse of already loaded values. The parameter $\delta$ depends on architectural parameters.
5. Use graph coloring to generate a virtual to logical register mapping.
6. Use the BRILA scheduler, described in Figure 7, for the corresponding architecture to produce a final schedule for the microkernel.

**Figure 8: Register allocator and scheduler in BRILA**

## 4.3 R($R_U \times R_U \times R_U$, BC)

Although Belady's algorithm minimizes the number of loads, it is not necessarily the best approach to register allocation on modern machines; for example, we can afford to execute more than the optimal number of loads from memory if they can be performed in parallel with each other or with computation[4]. Therefore, we decided to investigate an integrated approach to register allocation and scheduling [5, 17, 23]. Figure 8 briefly describes the algorithm we implemented in BRILA.

Both UltraSPARC IIIi and Itanium 2 are in-order architectures, and precise scheduling is extremely important for achieving high-performance. Figure 6(b) and 6(c) show that the BC strategy works better on these architectures than the other strategies discussed in this section. As we can see in Figure 6(b), the performance of this microkernel in isolation

[4]Belady invented his policy while investigating page replacement policies for virtual memory systems, and the algorithm is optimal in that context since page faults cannot be overlapped. Basic blocks, however, have both computation and memory accesses to schedule, and can overlap them to gain higher performance.

on the UltraSPARC is about 50% of peak for $R_U > 3$. The performance of the complete MMM is about 760 MFlops, or just about 38% of peak. On the Itanium 2 architecture, the performance of the microkernel in isolation is 93% of peak. Although this level of performance is not sustained in the complete MMM, Figure 4(c) shows that the complete MMM reaches about 3.8 GFlops or about 63% of peak.

The situation is more complex for the Power 5 since it is an out-of-order architecture and the hardware reorders instructions during execution. Figures 4(a) and 6(a) show that the Belady register allocation strategy (BB) performs better on Power 5 than the integrated graph coloring and scheduling approach (BC). Intuitively, this occurs because the out-of-order hardware schedules around stalls caused by the Belady register allocation.

## 4.4 Discussion

Our work on adding processor-consciousness to recursive MMM codes led us to the following conclusions.

- The microkernel is critical to overall performance of the recursive code. Producing a high-performance microkernel is a non-trivial job, and requires substantial programming effort.
- The performance of the program obtained by following the canonical recipe (recursive outer control structure and recursive microkernel) is substantially lower than the near-peak performance of highly optimized iterative codes in vendor BLAS. The best we were able to obtain was 63% of peak on the Itanium 2; on the UltraSPARC, performance was only 38% of peak.
- For generating the microkernel code, using Belady's algorithm followed by scheduling may not be optimal. Belady's algorithm minimizes the number of loads, but minimizing loads does not necessarily maximize performance. An integrated register allocation and scheduling approach appears to perform better.
- Most compilers we used did not do a good job with register allocation and scheduling for long basic blocks. This problem has been investigated before [5, 17, 23]. The situation is more muddied when processors perform register renaming and out-of-order instruction scheduling. The compiler community needs to pay more attention to this problem.

# 5. PROCESSOR-CONSCIOUS ITERATIVE CODES

We now discuss how processor-consciousness can be added to iterative codes.

## 5.1 Iterative microkernels

Most numerical linear algebra libraries as well as the ATLAS system [27] use iterative microkernels whose structure is shown pictorially in Figure 9. Unlike the recursive microkernels described in Section 4 that have a single degree of freedom $R_U$, the iterative microkernels have three degrees of freedom called $K_U$, $N_U$, and $M_U$. The microkernel loads a block of the $C$ matrix of size $M_U \times N_U$ (shown as a solid rectangle inside C in Figure 9) into registers, and then accumulates the results of performing a sequence of size $K_U$ of outer products between small column vectors of $A$ and small row vectors of $B$.

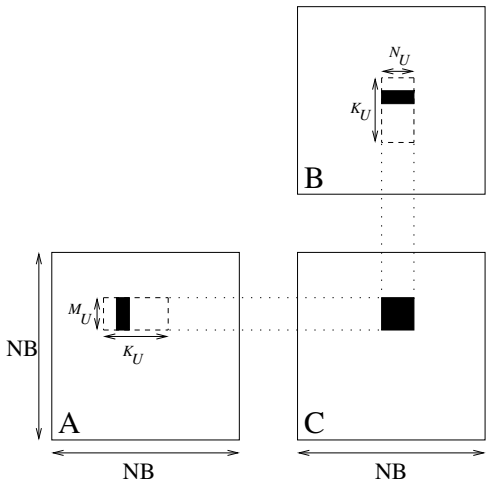



**Figure 9: Iterative microkernel and minikernel used in ATLAS**

Our iterative microkernels are generated by BRILA as follows.

1. Start with a simple $kji$ triply-nested loop for performing an MMM with dimensions $\langle K_U, N_U, M_U \rangle$ and unroll it completely to produce a sequence of $M_U \times N_U \times K_U$ FMAs.
2. Use the algorithm described in Figure 8 for register allocation and scheduling, starting with the sequence of FMAs generated above.

We examined the schedule of our microkernel and compared it to the structure of the ATLAS microkernel, which is shown in Figure 9. Both perform the computation instructions in the same order and keep the submatrix of C in registers at all times. Our BRILA compiler uses a description of the architecture to schedule the loads from A and B more precisely. ATLAS relies on the native compiler.

The iterative microkernel generated in this way has a number of advantages. For the $kji$ loop order, the number of required registers does not depend on the $K_U$ parameter [29, 27]. Thus we can optimize the values of $M_U$ and $N_U$ to make the working set of the microkernel fit in the register file. Then, we can optimize the value of $K_U$ to make the code of the microkernel fit in the instruction cache. In principle, we can generate recursive microkernels for non-square blocks, but their dimensions are not independent since each dimension affects both register allocation and instruction cache utilization.

Table 2 shows the performance of our iterative microkernels in isolation (also shown as a solid flat horizontal line in Figure 6(a-d)). We name the iterative microkernels with T for Tiled, the block size $M_U \times N_U \times K_U$ and the allocation - scheduling pair (BC or BB).

It can be seen that iterative microkernels perform substantially better than recursive microkernels on most architectures, obtaining close to peak performance on most of them.

## 5.2 Overall MMM Performance

To perform complete MMMs, the iterative microkernel is wrapped in an outer control structure consisting of a triply-

| Architecture | Micro-Kernel | Percent |
|---|---|---|
| Power 5 | R($8 \times 8 \times 8$, BB) | 58% |
| | T($4 \times 4 \times 120$, BC) | 98% |
| UltraSPARC IIIi | R($12 \times 12 \times 12$, BC) | 53% |
| | T($4 \times 4 \times 120$, BC) | 98% |
| Itanium 2 | R($9 \times 9 \times 9$, BC) | 93% |
| | T($8 \times 8 \times 12$, BC) | 94% |

**Table 2: Performance of the best microkernels in isolation.**

nested loop that invokes the iterative microkernel within its body. The resulting code is processor-conscious but not cache-conscious, and therefore has a working set of a matrix, a panel of another matrix and a block from the third matrix; because it uses a microkernel, it does provide register blocking. The experimental results are labelled with I, followed by the microkernel name from Table 2 in Figure 4(a-d).

On all four machines, the performance trends are similar. When the problem size is small, performance is great because the highly-tuned iterative microkernel obtains its inputs from the highest cache level. However, as the problem size increases, performance drops rapidly because there is no cache blocking. This can be seen most clearly on the Power 5. Performance of I T($4\times4\times120$, BC) is initially at 5.8 GFlops. When the working set of the iterative version becomes larger than the 1920KB L2 cache (for matrices of size $480 \times 480 \times 480$), performance drops to about 3.8 GFlops. Finally, when the working set of the iterative version becomes larger than the 36MB L3 cache (for matrices of size $2040 \times 2040 \times 2400$), performance drops further to about 2 GFlops, about 30% of peak.

## 5.3 Discussion

Table 2 shows that on a given architecture, iterative microkernels are larger in size than recursive microkernels. It is possible to produce larger iterative microkernels because of the decoupling of the problem dimensions: the size of the $K_U$ dimension is limited only by the capacity of the instruction cache, and is practically unlimited if a software-pipelined loop is introduced along $K_U$.

In summary, iterative microkernels outperform recursive microkernels by a wide margin on three out of four architectures we studied; performance of the recursive microkernel was close to that of the iterative microkernel only on the Itanium. Since overall MMM performance is bounded above by the performance of the microkernel, these results suggest that use of recursive microkernels is not recommended. Other researchers have recently come to the same conclusion [11]. In the rest of the experiments on matrix multiplication, we will therefore focus exclusively on iterative microkernels.

# 6. INCORPORATING CACHE BLOCKING

Without cache blocking, the performance advantages of the highly optimized iterative microkernels described in Section 5 are obtained only for small problem sizes; once the working set of the problem is larger than the capacity of the highest cache level, performance drops off. To sustain performance, it is necessary to block for the memory hierarchy.

In this section, we describe two ways of accomplishing this. The first approach is to wrap the iterative microkernel in a recursive outer control structure to perform approximate blocking. The second approach is to use iterative outer control structures and perform explicit cache tiling.

## 6.1 Recursive outer control structure

Figure 4 presents the complete MMM performance of the iterative microkernels within a recursive outer structure. The corresponding lines are labelled with R followed by the name of the microkernel from Table 2. On all four machines, performance stays more or less constant independent of the problem size, demonstrating that the recursive outer control structure is able to block approximately for all cache levels. The achieved performance is between 60% (on the UltraSPARC IIIi) and 75% (on the Power 5) of peak. While this is good, overall performance is still substantially less than the performance of the native BLAS on these machines. For example, on the Itanium, this approach gives roughly 4 GFlops, whereas vendor BLAS obtains almost 6 GFlops; on the Ultrasparc III, this approach obtains roughly 1.2 GFlops, whereas vendor BLAS gets close to 1.6 GFlops.

## 6.2 Blocked iterative outer control structure

These experiments suggest that if an iterative outer control structure is used, we may need to tile explicitly for all levels of the memory hierarchy. A different approach that leads to even better performance emerges if one studies the handcoded BLAS on various machines. On most machines, the handcoded BLAS libraries are proprietary, so we did not have access to them. However, the ATLAS distribution has hand-optimized codes for some of the machines in our study. The minikernels in these codes incorporate one level of cache tiling, and prefetching is done so that while one microkernel is executing, data for the next microkernel is prefetched from memory [26]. The resulting performance is very close to that of vendor BLAS on all machines (we do not show these performance lines in Figure 4 to avoid cluttering the figure).

To mimic this structure, we used the BRILA compiler to generate a *minikernel* composed of a loop nest wrapped around the $M_U \times N_U \times K_U$ iterative microkernel; the minikernel performs a matrix multiplication of size $N_B \times N_B \times N_B$ by invoking the iterative microkernel repeatedly, as shown in Figure 9. This is essentially the structure of the minikernel used in the ATLAS system [27].

For our experiments, we set $N_B$ to 120 on all machines. The rationale for choosing 120 as the cache tile size is the following. For the machines in our study, it is not necessary to tile for the L1 cache. On all machines we looked at there is enough concurrency in the optimized microkernel to cover the latency of L2 cache accesses. A tile size of 120 satisfies Inequality (5) for the L2 cache on all the machines in our study other than the IBM Blue Gene where it is satisfied for the L3 cache. In addition, it is advantageous to make the cache tile size some multiple of the register tile size to avoid performance problems that arise from "leftover" pieces within the cache tile that cannot be executed by the register tile. The number 120 is a composite number divisible by the sizes of the register tiles in our study. Therefore, our blocked iterative code provides a counterpoint to the recursive code since it uses a single cache tile size instead of a entire range of cache tile sizes.

As with the microkernels, this minikernel can then be used with either recursive or iterative outer control structures. The experimental results in Figure 4 show a number

of interesting points. The recursive and iterative outer control structures achieve almost identical performance for most problem sizes. For instance R T(120,4x4x120,BC) reaches nearly 6 GFlops on the Power 5 and maintains its performance through matrix sizes of $5000 \times 5000$. On the other hand, I T(120,4x4x120,BC) matches this performance until the matrices become too large to fit in the L2 cache; performance then falls off because there is no tiling for the L3 cache.

We have not yet implemented prefetching in BRILA, but for iterative minikernels, the memory access pattern is regular and predictable, so instructions that touch memory locations required for successive microkernels can be inserted into the computationally intensive code of earlier microkernels without performance penalty. However, it is not clear how one introduces prefetching into programs with a recursive outer control structure. Following the line of reasoning described in Section 2, we believe this is required to raise the level of performance of the recursive approach to that of the iterative approach. Whether prefetching can be done in some cache-oblivious manner remains to be seen.

## 6.3 Discussion

Our minikernel work led us to the following conclusions.

- Wrapping a recursive control structure around the iterative microkernel gives a program that performs reasonably well since it is able to block approximately for all levels of cache and block exactly for registers.
- If an iterative outer control structure is used, it is necessary to block for relevant levels of the memory hierarchy.
- To achieve performance competitive with hand-tuned kernels, minikernels need to do data prefetching. It is clear how to do this for an iterative outer control structure but it is not clear how to do this for a recursive outer control structure.

# 7. MATRIX TRANSPOSE

Matrix Transposition (MT) is different in behavior from MMM. It does $O(N^2)$ work on $O(N^2)$ data, so there is no algorithmic reuse, but it can benefit from exploiting spatial locality in data cache and data TLB. There are no multiply-add operations in the microkernel, but an important performance metric is the rate at which data is stored in memory.

We performed our experiments with out-of-place MT on the IBM Blue Gene architecture. We used standard row-major data layout and ran four different algorithms.

- Naïve Iterative - double nested loop;
- Naïve Recursive - divide-all-dimensions recursion;
- Optimized CO - following the CO recipe; divide-all-dimensions recursive outer structure with an optimized $32 \times 32$ microkernel; and
- Optimized CC - the ESSL vendor library.

To study the impact of conflict misses, we ran all experiments with the leading dimension equal to the size of the corresponding matrix (*exact*), as well as with a leading dimension slightly larger than the size of the corresponding matrix (*padded*). In Figure 10 we show our results for

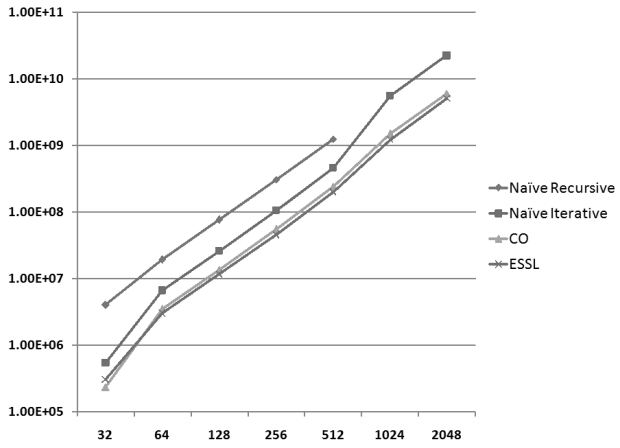



**Figure 10: Matrix Transpose runtime in cycles on IBM Blue Gene for all four implementations using square matrices (log scale).**

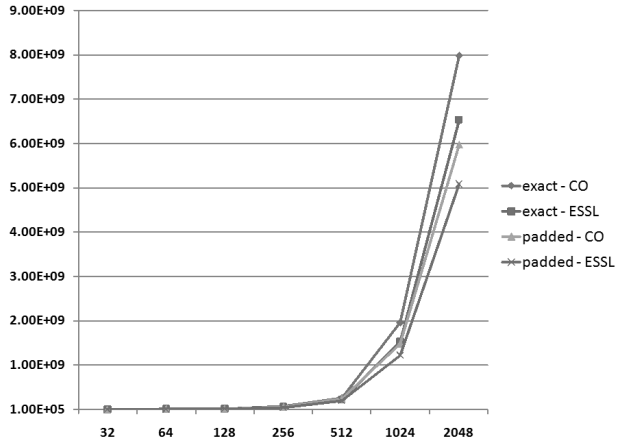



**Figure 11: Matrix Transpose runtime in cycles on IBM Blue Gene for padded and exact optimized CO and CC implementations using square matrices (linear scale).**

"padded" (note the log scale). The best performance is obtained by ESSL, followed by the optimized CO, Naïve Iterative, and finally the Naïve Recursive. For space reasons we do not show the graph for "exact", as it is very similar and the relative performance is preserved, though performance is smaller in absolute sense due to cache conflict misses.

Figure 11 shows the relative performance of the optimized CO and CC implementations for both "exact" and "padded" data (note the linear scale). The ESSL CC implementation on padded data is fastest. Optimized CO on padded data is about 20% slower. The experiments on "exact" data exhibit a similar relative gap.

These results are in line with the results from the MMM results discussed earlier: (i) naïve iterative and recursive implementations do not exhibit good performance; (ii) it is a substantial effort to produce a good microkernel for both optimized CO and optimized CC approaches, and (iii) optimized cache-conscious code usually performs significantly better than the optimized cache-oblivious code.

# 8. CONCLUSIONS AND FUTURE WORK

We began this paper by asking whether there was a price that cache-oblivious programs pay for the ability to adapt automatically to the memory hierarchy. The results in this paper provide a quantitative answer, in two parts.

First, our experiments show that high-performance codes must at least be *processor-conscious* in the sense that they require microkernels that are carefully optimized for the processor pipeline, registers and L1 I-cache. We also found that iterative microkernels for matrix-multiplication performed better than the recursive microkernels we produced using the methods prescribed in the literature. We believe this is because the iterative microkernels have more degrees of freedom that permit independent optimization for different processor resources, but this needs to be investigated.

Second, our experiments show that on current architectures, a single level of cache tiling and modest prefetching are adequate to generate high-performance matrix multiplication code. In particular, it is sufficient to tile approximately for the L2 cache on most current architectures, so adapting to all levels of the memory hierarchy by generating a range of tile sizes using divide-and-conquer appears to be unnecessary.

The results in this paper suggest several directions for future research.

- Are these conclusions valid for multicore architectures [9]?
- Can we produce better microkernels starting from the recursive code?
- Better register allocation and scheduling techniques are needed for long basic blocks.Using Belady's algorithm followed by scheduling is not necessarily optimal because minimizing the number of loads is not necessarily correlated to optimizing performance on architectures that support multiple outstanding loads and can overlap loads with computation.
- How do we integrate prefetching into cache-oblivious algorithms?
- The naïve recursive code described in Section 3 is I/O optimal, but delivers only 1% of peak on all architectures. Intuitively, the I/O complexity of a program describes only one dimension of its behavior, and focusing on I/O optimality alone may be misleading when it comes to overall performance. What models are appropriate for describing other dimensions of program behavior to obtain a comprehensive description of program performance?

Acknowledgements: We would like to thank Matteo Frigo and Gianfranco Bilardi for useful discussions.

# 9. REFERENCES

[1] Basic Linear Algebra Routines (BLAS). `http://www.netlib.org/blas`.

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

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition*. SIAM, Philadelphia, 1995.

[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Integrated instruction scheduling and register allocation techniques. In *LCPC '98*, pages 247–262, London, UK, 1999. Springer-Verlag.

[6] Gianfranco Bilardi, 2005. Personal communication.

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

[8] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *PLDI*, pages 53–65, 1990.

[9] Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.

[10] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. Recursive array layouts and fast parallel matrix multiplication. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 222–231, 1999.

[11] Rezaul Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.

[12] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *PLDI*, 1995.

[13] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

[14] J. J. Dongarra, F. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, 1984.

[15] Matteo Frigo, 2005. Personal communication.

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

[17] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ICS '88*, pages 442–452, New York, NY, USA, 1988. ACM Press.

[18] Jia Guo, María Jesús Garzarán, and David Padua. The power of Belady's algorithm in register allocation for long basic blocks. In *Proc. 16th International Workshop in Languages and Parallel Computing*, pages 374–390, 2003.

[19] Fred Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.

[20] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. of the thirteenth annual ACM symposium on Theory of computing*, pages 326–333, 1981.

[21] Piyush Kumar. Cache-oblivious algorithms. In *Lecture Notes in Computer Science 2625*. Springer-Verlag, 1998.

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

[23] Cindy Norris and Lori L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *MICRO 28*, pages 169–179, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

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

[25] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External memory algorithms*. American Mathematical Society, Boston, MA, 1999.

[26] Clint Whaley. personal communication, 2005.

[27] 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.

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

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