Thread Scheduling for Cache Locality

James Philbin and Jan Edler
NEC Research Institute, 4 Independence Way, Princeton, NJ 08540

Otto J. Anshus
Department of Computer Science, Institute of Mathematical and Physical Sciences,
University of Tromso, N-9037 Tromso, Norway

Craig C. Douglas
IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598-0218; and
Department of Computer Science, Yale University, P.O. Box 208285, New Haven, CT 06520-8285

Kai Li
Department of Computer Science, Princeton University, Princeton, NJ 08540

Abstract

This paper describes a method to improve the cache locality of sequential programs by scheduling fine-grained threads. The algorithm relies upon hints provided at the time of thread creation to determine a thread execution order likely to reduce cache misses. This technique may be particularly valuable when compiler-directed tiling is not feasible. Experiments with several application programs, on two systems with different cache structures, show that our thread scheduling method can improve program performance by reducing second-level cache misses.

1 Introduction

The performance gap between processors and memory has widened in the last few years. Since the beginning of the 1980s, processor performance has been improving at the rate of about 50% per year, whereas the access time of DRAMs has improved at approximately 10% per year [19, 40]. To compensate for this gap, today's computer systems commonly have two levels of cache memory. However, the cost of a second-level cache miss is now, typically, more than 100 instructions. For example, on an SGI Indigo2 (R4400) with a peak execution rate of 200 MIPS, a main memory access takes 1.17 microseconds [32]. As the memory access penalty increases, application performance becomes more sensitive to cache misses. This is especially true for applications with data sets larger than the cache size that do not exhibit a high degree of memory reference locality.

One attractive way to ameliorate the processor/memory performance gap is to improve the data locality of applications. Tiling (also called blocking), a well-known software technique [1, 12, 18, 29], achieves this goal by restructuring a program to re-use certain blocks of data that fit in the cache. Tiling can reduce cache misses and can be applied to any level of the memory hierarchy, including virtual memory, caches, and registers. It can be done either automatically, by a compiler, or manually, by the programmer.

This paper describes an alternative technique to improve locality using fine-grained threads. The technique can be simpler than hand tiling and can generate better cache behavior than untiled code. Threads have traditionally been used to express concurrency and to exploit parallelism on multiprocessors. We use threads in *sequential* programs to create separate units of computation, without data dependencies, which can then be reordered to improve data locality. Avoiding a secondary cache miss on current machines saves 100 or so instruction times. This more than offsets the cost of creating, scheduling, and running a lightweight thread [10, 17], assuming thread creation doesn't cause cache misses. If the threads are scheduled in an order that results in many fewer cache misses, the resulting program will run faster.

The scheduling algorithm uses address information, provided by the programmer on thread creation, as hints for determining an execution ordering that increases data locality. We have designed and implemented a user-level thread package that relies upon our scheduling algorithm to reduce second-level cache misses. The thread package supports very fine-grained, "run-to-completion" threads.

Tiling will normally yield better performance than our technique when it can be applied. However, there are several cases in which compiler directed tiling is infeasible. (1) The control or data flow complexity of a program may preclude static analysis, e.g., data might be allocated dynamically or accessed indirectly. (2) Physically addressed second level caches are more difficult to analyze, because of virtual memory effects [27], which may prevent the compiler from doing an effective job. (3) Many commercial compilers, particularly on PC's, do not perform tiling, or tile only simple programs. Without compiler help, the programmer who desires good performance must manually tile the algorithm — a complex task. Our technique provides a potentially simpler alternative, while still achieving most of the performance benefits of tiling. It is conceivable that this technique could be used by a compiler, but that investigation is beyond the scope of this paper.

We have experimented with the thread package using several application programs, including matrix multiplication, a partial differential equation solver, a successive over-relaxation kernel, and an N–body program. We replaced certain inner loops with fine-grained threads, and then ran the transformed programs on an SGI Power Indigo2 with an R8000 CPU and an SGI Indigo2 IMPACT with an R10000. Our results show that the thread scheduling method can significantly improve performance by reducing second-level cache misses. In particular, it can improve the performance of programs that have little or no compile time information about memory references.

2 Locality Scheduling

Our approach uses fine-grained threads to decompose a program and schedules these threads so as to improve the program's data locality.

2.1 Fine-Grained Threads

It seemed to us that it would be easy for a programmer to transform a sequential program into a threaded program, with a thread system supporting very fine-grained threads. By fine-grained, we mean that the overhead of thread primitives is small in both time and space requirements. With fine-grained threads, for example, one can substitute a thread for the inner-most loop of a program that may be causing second-level cache misses.

Consider a matrix multiplication example, $C = A \times B$, where A, B, and C are all n by n matrices. In order to improve locality, A is transposed before and after the computation. One straightforward implementation uses nested loops:

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } n \\
\quad \textbf{for } j = 1 \textbf{ to } n \\
\quad\quad C[i,j] = 0; \\
\quad\quad \textbf{for } k = 1 \textbf{ to } n \\
\quad\quad\quad C[i,j] = C[i,j] + A[k,i] * B[k,j];
\end{array}
$$

The inner-most loop computes the dot product of two n-element vectors. Note, we are using this example to illustrate the thread system. A good compiler can tile this expression easily.

With fine-grained threads, we can simply replace the dot-product loop with a thread:

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } n \\
\quad \textbf{for } j = 1 \textbf{ to } n \\
\quad\quad Fork(DotProduct, i, j); \\
RunThreads();
\end{array}
$$

$$
\begin{array}{l}
DotProduct(i,j): \\
\quad C[i,j] = 0; \\
\quad \textbf{for } k = 1 \textbf{ to } n \\
\quad\quad C[i,j] = C[i,j] + A[k,i] * B[k,j];
\end{array}
$$

Where $Fork$ creates and schedules a thread that will compute the specified dot product, and $RunThreads$ runs each thread in some order determined by the scheduling algorithm.

This example shows how such transformations can be performed by hand. However, fine-grained threads present challenges in implementation. In order for our approach to be effective, the total cost associated with threading must be significantly less than the cost of the cache misses eliminated. This requires that the scheduling algorithm make intelligent decisions that improve the program's locality.

2.2 The Scheduling Problem

In order to schedule threads for data locality, the scheduler must be informed about the memory references made by the threads. To study the problem, let us assume that each thread references k pieces of data during its execution and that the addresses of these k pieces are known to the scheduler. Let us further assume that all threads are independent.

Thread t_i is denoted by $t_i(a_{i1}, \ldots, a_{ik})$ where a_{ij} is the address of the j^{th} piece of data referenced by thread t_i during its execution. Thus, if n threads are executed in some order $t_1, t_2, \ldots, t_n$, they can be represented by the permutation:

$$
\begin{array}{l}
t_1(a_{11}, \ldots, a_{1k}), \\
t_2(a_{21}, \ldots, a_{2k}), \\
\quad\quad \vdots \\
t_n(a_{n1}, \ldots, a_{nk}).
\end{array}
$$

The goal in scheduling n threads for cache locality is to find a permutation that minimizes the number of cache misses. We can view such a problem as a k-dimensional geometry problem [11]. A thread $t_i(a_{i1}, \cdots, a_{ik})$ is a point in the k-dimensional space where the coordinates of the point are $(a_{i1}, \cdots, a_{ik})$. The problem is then equivalent to finding a tour of the thread points in the space that satisfies the requirement of minimizing cache misses. Such requirements depend on the cache's design.

It may not be possible to obtain a precise definition for a realistic system. Second-level caches are often physically indexed, while the addresses associated with the threads are virtual addresses. Past research has shown that the virtual-to-physical memory mapping maintained by the virtual memory system can significantly affect second-level cache behavior [8]. Knowledge of the current virtual-to-physical memory mapping and of the page swapping algorithm for the virtual memory would be needed to derive an optimal solution.

Furthermore, the complexity of finding a tour in a space with any global conditions based on the coordinates is likely to be expensive. A good example is the traveling salesman problem where the condition is to find the shortest global path. Instead of trying to define the problem precisely and to find an optimal algorithm that solves it, we focus on an efficient heuristic algorithm.

2.3 Our Algorithm

The main idea of our algorithm is to schedule threads in "bins" so that when the threads in a bin are run they will not cause many capacity cache misses. We use the k addresses associated with a thread as *hints* to the scheduler. We first describe an algorithm that uses two addresses associated with each thread and then explain how to extend the algorithm to use k addresses.

With $k = 2$ addresses, we constrain the general problem. The k addresses $(a_{i1}, \ldots, a_{ik})$, which act as hints to the scheduler, become two (a_{i1}, a_{i2}). Intuitively, these might be the two largest objects referenced by the thread or the two objects most frequently referenced by the thread.

Thus constrained, the scheduling problem then becomes finding a tour of points in a two-dimensional plane as shown in Figure 1, where each thread is represented as a point in the plane with coordinates defined by the two address hints. To minimize cache misses, the scheduling algorithm must find a tour that has a "cluster" property, i.e., threads that have the same or similar hints should be clustered together in the tour.

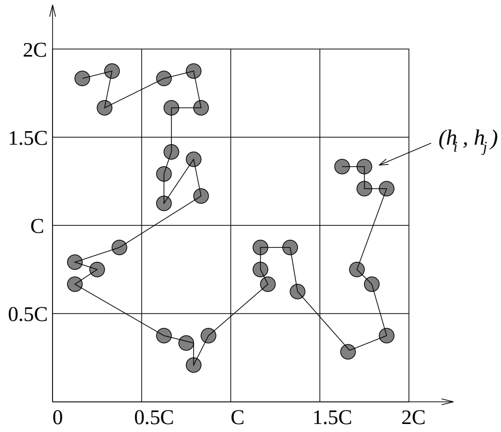

Figure 1: Scheduling threads in a 2-D plane.

The key idea of the algorithm is to use a two-dimensional block to determine the data locality of threads. Threads with data in the same block will be placed in the same bin so that they will be scheduled together. We divide the two-dimensional plane into equally sized blocks. The two address hints (h_i, h_j) of a thread are used as coordinates to place it into a bin. In other words, each block covers two pieces of user memory, one from each dimension. The insight is that if the sum of the two dimensions of the block is less than the cache size C and if the threads falling into the same block are scheduled together, then the execution of the threads will not cause many cache misses. Thus, we make the size of each dimension of a block be less than or equal to one half of the cache size.

Our scheduling algorithm uses bins to schedule threads. When a thread is created, its address hints are used to index into the scheduling plane to determine a bin for scheduling. Scheduling involves traversing the bins along some path, preferably the shortest one. For each non-empty bin, the scheduler runs all threads it contains in some order. If the scheduler focuses only on minimizing the cache misses of the largest cache (for example, the second-level cache in a system with two levels of caches), then the scheduling order of threads in the same bin can be arbitrary.

We implement this algorithm by hashing the hint addresses to place threads into a fixed number of bins. Furthermore, threads with address hints (h_i, h_j) and (h_j, h_i) can be placed in the same bin, since they reference the same pieces of data. An implementation can take advantage of this property to reduce the number of bins by 50%.

The algorithm for k addresses is a simple extension to the algorithm above. Instead of using a two-dimensional block in a two-dimensional plane to determine the locality of threads, the general algorithm uses a k-dimensional block in a k-dimensional space. The sizes of the block dimensions should be set such that the sum of the k dimensions of the block is less than or equal to the cache size. Threads with data in the same block will be placed into the same bin.

2.4 An Example

To show how the thread scheduler works, let us consider a simple example of a 4×4 matrix multiplication ($C = A \times B$) program using the threaded algorithm presented in section 2.1. The inner-loop forks off threads, each of which computes a dot product, in the following order:

$$t_1(a_1, b_1), t_2(a_1, b_2), t_3(a_1, b_3), t_4(a_1, b_4),$$
$$t_5(a_2, b_1), t_6(a_2, b_2), t_7(a_2, b_3), t_8(a_2, b_4),$$
$$t_9(a_3, b_1), t_{10}(a_3, b_2), t_{11}(a_3, b_3), t_{12}(a_3, b_4),$$
$$t_{13}(a_4, b_1), t_{14}(a_4, b_2), t_{15}(a_4, b_3), t_{16}(a_4, b_4)$$

where a_i and b_j are the i-th vector of matrix A and j-th vector of matrix B respectively.

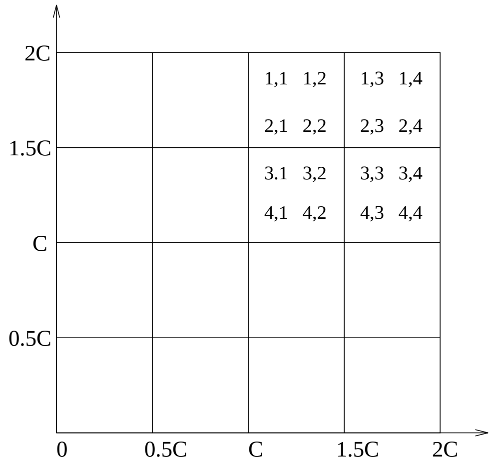

Figure 2: An example of scheduling threads.

Suppose the cache holds only four vectors and the dimension size of a block in the 2-D scheduling plane is one half the cache size. Then in one possible mapping (depending on the hashing function) threads fall into four blocks of the schedule plane, as shown in Figure 2, where each i, j pair represents a thread that computes the dot product for a_i and b_j.

Suppose each block maps to a bin, then our algorithm puts the threads in each block into the same bin:

$$\begin{aligned} bin_1 &= \{t_1(a_1, b_1), t_2(a_1, b_2), t_5(a_2, b_1), t_6(a_2, b_2)\} \\ bin_2 &= \{t_3(a_1, b_3), t_4(a_1, b_4), t_7(a_2, b_3), t_8(a_2, b_4)\} \\ bin_3 &= \{t_9(a_3, b_1), t_{10}(a_3, b_2), t_{13}(a_3, b_3), t_{14}(a_3, b_4)\} \\ bin_4 &= \{t_{11}(a_4, b_1), t_{12}(a_4, b_2), t_{15}(a_4, b_3), t_{16}(a_4, b_4)\} \end{aligned}$$

When the threads are run, the bins will be traversed in the above order, with all threads in each bin being executed before moving to the next bin. The scheduler thus reorders the dot-products in such a way that the execution of the threads in each bin will not cause capacity cache misses. The scheduling order exhibits similar locality to that provided by course grain tiling either by hand or by a compiler.

3 Thread Package

This section describes a thread package for data locality scheduling. The thread package consists of a thread scheduler that conforms to the algorithm described in the previous section, with primitive operations optimized to support fine-grained, user-level threads.

In principle, a general purpose thread package [5, 7, 9, 13, 15] could be adapted to schedule threads according to memory access hints. While significant variation exists among current thread packages, they all include support for synchronization, including context switching and thread state saving. Many thread packages include additional features, such as preemptive scheduling, asynchronous signalling mechanisms, and private data areas; such features can be valuable in some contexts, expanding the class of problems for which a package is applicable. However, our design for locality scheduling keeps the thread package simple, making low-overhead the most important goal.

We have implemented a special-purpose thread system for locality scheduling that features a minimal user interface and very low overhead. Because our threads always "run-to-completion," and have no support for blocking and resuming execution, the system is implemented without resorting to assembly language. In fact, the essential functionality of the library is embodied in about 525 lines of C, including both C and Fortran-callable interfaces. However, this does not imply that it is completely free of machine and configuration dependencies. In particular, as explained above, the cache size is an important parameter of the scheduling algorithm.

Our thread package implements the scheduling algorithm for the three-dimensional case, although it is quite easy to extend it to higher dimensional cases. It uses three-dimensional blocks to determine data locality to place threads into bins.

3.1 User Interface

The basic user interface currently consists of only three calls:

- $th\_init(blocksize, hashsize)$

  Set block size and hash table size. This function can be called more than once to change those sizes. The configuration-dependent default value can be selected by passing 0.

- $th\_fork(f, arg_1, arg_2, hint_1, hint_2, hint_3)$

  Create and schedule a thread to call $f(arg_1, arg_2)$. The $hint_1$, $hint_2$ and $hint_3$ are memory addresses used as scheduling hints. For the two-dimensional case, $hint_3$ will be 0. Similarly, for one-dimensional case, both $hint_2$ and $hint_3$ will be 0.

- $th\_run(keep)$

  Run all threads that have been scheduled, by th_fork, and then return. The thread specifications will be destroyed if $keep$ is 0, or saved to allow re-execution otherwise.

The user interface functions do not return any value. There are no handles or other identifiers for threads, and no operations that act on them individually. Error handling, as necessary, is entirely internal, reflecting the current research use of the system in batch-oriented scientific programs.

3.2 Implementation

The thread package is implemented with four basic data structures: thread group, bin, hash table, and ready list. Figure 3 shows the relationships of these data structures.

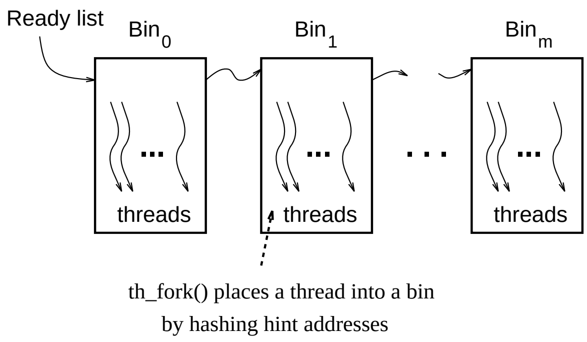

Figure 3: Relationships of threads, bins, and ready list.

The thread group data structure represents a number of threads within a bin; by grouping threads together in this way, amortization reduces the cost of thread structure management. Since the thread package supports only non-preemptive scheduling, the data required to represent each thread is very simple: a void function pointer and the two arguments arg_1 and arg_2 supplied by the user to th_fork. The thread group consists of an array of these structures plus an integer to count the number of threads actually in the group and a pointer to the next thread group in the bin.

Since the thread package has no preemptive scheduling, there is only one stack used by the program. As each thread terminates, the stack is given to the next thread by the scheduler.

The bin data structure contains three link fields and a search key. The first field links bins that fall into the same hash block. The hashing function guarantees that threads falling into the same block in the scheduling plane (as described in the previous section) will be placed in the same bin. The second link field chains all threads groups associated with the same scheduling bin. The ready list uses the third link field for scheduling.

The hash table organizes the bins. Hash collisions are resolved by chaining, and the table is simply a three-dimensional array of pointers to bins. An application can change the hash function by setting its own block size and hash table size.

The ready list contains threads that are ready to run. The ready list is a simple linked list containing all allocated bins. Each time a new bin is allocated, it is added to the end of this list. The scheduler does not allocate a bin in the hash table until it schedules the first thread in it. When *th_run* is called, the ready list is traversed, in order, to find and execute all threads.

The mapping from hints to blocks and the hash function are reconfigurable. The default dimension sizes of the block are set such that their sum are the same as the second-level cache size. The hashing function hashes all hints as indices to a bin. The default hashing function simply performs a shift and a mask operation on each hint.

4 Experiments

Our experiments were designed to measure the overhead of the thread package and to measure the performance of applications threaded for cache locality. We experimented with the thread package on two computer systems: SGI Power Indigo2 and SGI Indigo2 IMPACT. We conducted experiments on two different machines to examine the effect on the thread package of different system performance characteristics. Both systems were equipped with the same amount of memory (128MB), easily accommodating our tests without any paging.

The SGI Power Indigo2 has a 75 MHz MIPS R8000 CPU, separate 16KB first-level instruction and data caches, and a unified 2 MB 4-way set associative second-level cache. The line sizes are 32 and a 128 bytes for the first and second level caches, respectively. The R8000 has two floating point units, two integer units, and two memory load/store units [23]. It is capable of executing up to four instructions each clock cycle. Its SPECint92 performance is 113 and SPECfp92 is 269. (Estimated SPECint95 is 2.8 and SPECfp95 is 8.9.)

The SGI Indigo2 IMPACT is a newer system, with a 195 MHz MIPS R10000 CPU, separate 32KB 2-way set associative first-level instruction and data caches, and a unified 1 MB 2-way set associative second-level cache. The line sizes are: 64 bytes, 32 bytes, and a 128 bytes, respectively (I, D, and L2 caches). The R10000 is a 4-way superscalar CPU with two integer pipelines, two floating point pipelines, one load/store pipeline, and out of order execution [33]. Its SPECint95 performance is 8.9 and SPECfp95 is 12.5.

We first measured the overhead of thread primitives and then measured the performance of four applications. The first three applications are written in Fortran (column major layout), and the last is written in C (row major layout). Either layout works with our scheduler. In general, for each application, we conducted the following sets of runs: the best unthreaded/untiled version, the compiler tiled and/or hand tiled or cache conscious version (when available), and the threaded version. All times reported are in CPU seconds.

In order to understand our results better, we performed cache simulations, on the SGI Power Indigo2, using address reference traces taken from the same executable binary programs that were used to produce the timing results. The address traces were generated by Pixie [37], and processed by a modified DineroIII cache simulator [20, 21]. Our modifications to DineroIII allow it to process traces containing 2^{31} or more references, to classify misses as compulsory, capacity, or conflict in a single run, and to further reduce simulation time by directly reading the binary Pixie trace output. The results presented exclude program initialization costs.

The rest of this section describes the benchmarks, reports the performance and cache simulation numbers, and analyzes the results.

4.1 Micro Benchmarks

Table 1 reports the overhead to fork (create and schedule) and run (execute and terminate) a null thread on both machines. The total overhead associated with the thread is also shown. The last row of the table shows the cost of an L2 cache miss on each machine [24, 34, 35]

	R8000	R10000
Fork	1.38	0.95
Run	0.22	0.14
Total	1.60	1.09
L2 Miss	1.06	0.85

Table 1: Thread overhead in microseconds.

The thread overheads were calculated using a simple loop that created 1,048,576 threads to call the null procedure and then ran them. The threads were evenly distributed across the scheduling plane.

4.2 Matrix Multiply

Matrix multiplication is a simple example to show the differences among different methods. A matrix multiplication can be computed in several ways. The most common sequential method is:

for $j = 1$ **to** n
  for $k = 1$ **to** n
    for $i = 1$ **to** n
      $C[i,j] = C[i,j] + A[i,k] * B[k,j];$

This method interchanges the 3 loops to reduce cache misses (column-major storage order assumed). Since $B[k, j]$ is not changed in the inner-most loop, it can be lifted out and put in a register in the middle loop. Thus, the minimum number of memory references for each iteration of the inner-most loop is two loads and one store. We call this version **interchanged**.

Another cache-conscious algorithm transposes matrix A before and after the multiplication so that the dot-product loop will access data elements from two sequentially stored vectors. The loops are:

for $i = 1$ **to** n
  for $j = 1$ **to** n
    for $k = 1$ **to** n
      $C[i, j] = C[i, j] + A[k, i] * B[k, j];$

In this program, the address of $C[i, j]$ is not changed in the inner-most loop. This enables a compiler or a programmer to use a register in the inner-most loop and only store the value back to memory when the loop finishes. Thus, the minimum number of memory instructions in each iteration of the inner-most loop is two loads. Obviously, this approach must transpose matrix A twice. However, since the complexity of a transpose is an order of magnitude less than the matrix multiply, the overhead of transposes is small. We call this version **transposed**.

The threaded version is produced by substituting the inner-most loop with a thread:

for $i = 1$ **to** n
  for $j = 1$ **to** n
    $th\_fork(DotProduct, i, j, A[1, i], B[1, j]);$
$th\_run(0);$

$DotProduct(i, j):$
  for $k = 1$ **to** n
    $C[i, j] = C[i, j] + A[k, i] * B[k, j];$

We call this version **threaded**.

We have looked at several compiler tiled matrix multiplication programs. We tiled both **interchanged** and **transposed** with the KAP compiler of Kuck & Associates, Inc. for the R8000, and with the SGI Version 7.0 compiler for the R10000.

Table 2 shows the performance of these matrix multiplications ($n = 1024$). All programs were compiled with the SGI Fortran compiler using its `-O3`, `-mips4`, and `-n32` switches. For the tiled programs on the R8000, we first invoke the KAP compiler to perform tiling and then use the SGI compiler to compile the programs. On the R10000 we use SGI's version 7.0 Fortran compiler which tiles the program without using KAP.

The performance numbers for the transposed and threaded versions include the overhead of two matrix transpose operations. The threaded version uses a 2-D scheduling plane to determine the locality of threads and sets the block size to be one half of the second-level cache size on both systems. On the R8000 (with 2MB cache), the threaded version creates 1,048,576 threads distributed in 81 bins for an average of 12,945 threads per bin. The distribution of the threads in the bins was quite uniform.

	R8000	R10000
Interchanged	102.98	36.63
Transposed	95.06	32.96
Tiled interchanged	16.61	12.24
Tiled transposed	19.73	18.71
Threaded	20.32	16.85

Table 2: Matrix multiply performance in seconds.

Although the threaded version improves the performance of the untiled version by a factor of 5 on the R8000 and more than a factor of 2 on the R10000, the compiler tiled version enhances performance by factors of 6 and 3 respectively. The compiled version has better performance because it can tile the registers, first-level cache and second-level cache, whereas the threaded method only reduces misses in the second-level cache.

R8000	Untiled	Tiled	Threaded
I fetches	5,388,645	2,184,458	3,929,858
D references	3,222,274	728,256	2,193,690
L1 misses	408,756	215,652	414,741
rate	4.8	7.4	6.8
L2 misses	68,225	738	1,872
rate	4.6	0.3	0.4
L2 compulsory	199	200	299
L2 capacity	68,025	528	1,311
L2 conflict	0	10	262

Table 3: Matrix multiply memory references and cache misses in thousands.

Table 3 shows the cache simulation results for the R8000 system. It includes the results for the untiled interchanged, compiler tiled interchanged, and threaded versions. The table shows that capacity misses dominate the L2 cache misses and that both the tiled version and threaded versions can reduce most of the misses.

The tiled version reduces the number of instruction and data references by 3,204M and 2,494M respectively. Such a significant reduction is due to differences in code generation. The forms of the inner loops are somewhat different: the untiled interchanged version has to store partial sums, which the threaded version (being based on a transposed algorithm) does not, this directly reduces the number of stores and also allows the compiler to generate a tighter loop with fewer instructions per result. The transformations performed by KAP include tiling for the registers, which allows an even greater reduction in memory references through re-use and produces even tighter inner loops. Although the SGI compiler (with the same optimization switch) unrolls the inner loops for all three versions, the unrolled results are quite different for the reasons above: 10 instructions with 2 multiply-adds, 4 loads, 2 stores, 1 integer add, and 1 branch for the untiled version; 18 instructions with 9 multiply-adds, 6 loads, 2 integer adds, and 1 branch for the tiled version;

and 14 instructions with 4 multiply-adds, 8 loads, 1 integer add, and 1 branch for the threaded version. If we ignore the rest of the programs and consider only these inner loops, we can account for nearly all instructions used to produce the necessary 2^{30} multiplications (that is 5,368,709, 2,147,484, and 3,758,096 thousand instructions, respectively).

Tiling for the registers and L1 cache not only reduces the instruction count and total memory references, but it also significantly reduces the number of L1 cache misses, by about 193M, and L2 cache misses, by 67.5M.

The untiled version of the program is very memory intensive; we are not able to figure out exactly where the time goes by simply making crude assumptions. On average, there is one data memory reference for every three instructions and there is about one L1 cache miss for every ten instructions. The R8000 CPU has two integer units, two floating-point units, and two load/store units, and the processor can execute up to four instructions per cycle. It is difficult to make accurate assumptions about the pressure on the L1 cache without detailed CPU pipeline simulations. If we simply assume no L1 cache stalls and assume that overhead of an L1 and L2 cache miss is 7 cycles [23] and 1.06 microseconds respectively, then the estimated time saved would be about 83 seconds. The difference between such a crude estimate and the actual time saved (87 seconds) is only about 4 seconds. But reducing 3,204M instructions and 2,494 data references would take much more than 4 seconds. It would require a detailed R8000 CPU and Power Indigo2 memory system simulation to fully understand the performance implications.

The threaded version reduces the number of instruction and data references by 1,459M and 1,029M respectively. This is also due to code generation of inner loops. Since the threaded version is based on a transposed algorithm, it does not require storing partial sums, so the inner loop requires fewer instructions and fewer data stores. Unlike the tiled version, the threaded version increases the number of L1 cache misses by about 6M while reduces the L2 cache misses by about 66.4M. Most of the L2 misses avoided were capacity misses. Like the tiled version, this is a memory intensive program. With the crude assumptions, the threaded version would save about 69 seconds in L1 and L2 cache misses. Again, this program is also a memory intensive one; it would require detailed CPU pipeline and memory system simulations to fully understand where the time goes.

4.3 Linear Algebra Solvers

In this section we experiment with two iterative solvers. The first is meant to be nested inside a multigrid partial differential equation (PDE) solver. The second is a standard test case from the compiler community.

We experimented with three versions of the programs: regular implementation, cache-conscious version, and the threaded version.

Linear elliptic PDE's with boundary conditions can be solved after standard discretization methods (e.g., finite elements or differences) have been applied. A sequence of sets of sparse linear equations are solved:

$$A^{(i)}u^{(i)} = b^{(i)}, \quad i \geq 1.$$

When multigrid is used, $i > 1$; otherwise $i = 1$. The primary component in multigrid is the iterative method used to approximately solve each problem. While interpolation uses some time, it is a small percentage of the overall running time.

For an iterative method on a single grid, the typical algorithm has the form:

for $i_1 = 1$ **to** $iters$
  $action_1$
  $\vdots$
  $action_k$

In practical multigrid solvers, $iters \leq 5$.

Consider the following example of an iterative method to solve Laplace's equation on a rectangle with a uniform mesh:

for $i_1 = 1$ **to** $iters$
  $rb = mod(nx, 2)$
  for $krb = 1$ **to** 2
    for $i_3 = 1$ **to** ny
      for $i_2 = rb + 1$ to nx **by** 2
        $u[i_2, i_3] = \frac{1}{4}(b[i_2, i_3] - u[i_2 - 1, i_3] - u[i_2 + 1, i_3]$
          $-u[i_2, i_3 - 1] - u[i_2, i_3 + 1]);$
    $rb = 1 - rb;$

for $i_3 = 1$ **to** ny
  for $i_2 = 1$ **to** nx
    $r[i_2, i_3] = b[i_2, i_3] - 4u[i_2 i_3] - u[i_2 - 1, i_3]$
      $-u[i_2 + 1, i_3] - u[i_2, i_3 - 1] - u[i_2, i_3 + 1];$

This is just the red-black ordered Gauss-Seidel relaxation method with the residuals calculated afterwards. After the last iteration, the residuals would normally be projected onto a smaller grid's right hand side or the approximate solution would be interpolated into and added to a larger grid's approximate solution.

Neither the KAP compiler nor the SGI compiler can do any tiling transformations to improve the performance of this program. To take better advantage of the caches, one needs to convert the algorithm above into an algorithm that is cache conscious [14]. This method relaxes the $i_3 = 1$ line on the red points first. Then, for lines $1 < i_3 \leq ny$, the red points can be relaxed on line j, followed by the black points on line $i_3 - 1$. Finally, the black points can be relaxed on line j. The residual is calculated along with the black points where possible.

The threaded approach is similar. The block, which computes red points on one line and black ones on the trailing line, can be used as the thread. Hence there are $ny+1$ threads to do the work each iteration.

Both the cache conscious and threaded approaches result in the data passing through the cache $iters$ times instead of $2 \times iters + 1$ times.

	R8000	R10000
Regular	9.48	7.80
Cache-conscious	5.21	5.21
Threaded	7.24	4.98

Table 4: PDE performance in seconds.

Table 4 contains a summary of the CPU time to compute 5 iterations of u followed by the residual, for a problem size of 2049. The choice of 5 is motivated by what people routinely use in multigrid solvers.

The cache-conscious method is up to 45% faster than the regular method.

The threaded method creates one thread for each element of the result matrix, i.e., 2049 threads. The performance of the threaded method on the R8000 falls about half way between the regular and cache conscious methods. However, on the R10000 the threaded method is slightly faster then the cache conscious method, which shows no speedup over the R8000 version. It is not clear to us why the cache conscious algorithm did not speed up on the R10000. There are no tiled results in the table because neither the KAP nor the SGI compiler can tile this algorithm.

The threaded version is programmed with a specific ordering (red-black) which determines when an element of u is updated. This is somewhat more complicated than when any ordering is allowed. For the "any ordering" case, the threaded implementation is both simple and easily verified to be correct. However, both of these cases are trivial in comparison to the cache conscious version of the red-black ordering code.

R8000	Regular	Cache Conscious	Threaded
I fetches	303,686	277,622	283,467
D references	126,044	122,598	126,385
L1 misses	80,767	85,040	94,516
rate	18.8	21.2	23.1
L2 misses	6,038	2,888	3,415
rate	5.7	2.6	2.9
L2 compulsory	788	788	789
L2 capacity	5,251	2,100	2,627
L2 conflict	0	0	0

Table 5: PDE: cache misses in thousands.

Table 5 reports the data for the PDE cache simulation on the R8000. The problem size simulated is 2,049, just as in the execution experiment. Most of the L2 cache misses of the regular program are due to capacity misses. The cache-conscious version can avoid about 60% of the capacity misses while the threaded versions avoids about 50%.

The cache-conscious version has 26M instruction references and 3.4M data references less than the regular version because it uses a different algorithm. On the other hand, the cache-conscious version has 4.3M more L1 cache misses than the original, but it reduces L2 cache misses by about 3.2M. If we perform a crude analysis by assuming that each instruction takes a single cycle and that the L1 and L2 cache miss overheads are 7 cycles [23] and 1.06 microseconds, respectively, then the total time saved would be about 3.5 seconds, not so far from the actual time saved (4.3 seconds).

The threaded program has 20.2M instruction references less than the original program, but it has 345K more data references. The threaded program increases L1 cache misses by about 13.7M while reducing the L2 cache misses by about 2.6M. With the same crude analysis, the total estimated time saved is about 1.9 seconds, which is close to the actual time saved (2.2 seconds).

Although the crude analysis does not take many things into account, it offers some implications why the cache-conscious version outperforms the threaded version and why they both outperform the original program.

The second example presented in this section is a standard test case from the compiler community [29]. It is similar to the first example, but has some subtle differences. First, it uses less data. Second, the ordering of the updates is quite different. Third, it performs many more iterations (30). It is designed to solve a problem, not just accelerate another solver.

The successive over-relaxation (SOR) is an iterative method for solving large linear systems that are symmetric, positive definite. It is an extension of the Gauss-Seidel method. The SOR Nest, with t iterations is as follows:

$$
\begin{aligned}
&\textbf{for } i_1 = 1 \textbf{ to } t \\
&\quad \textbf{for } i_2 = 1 \textbf{ to } n-1 \\
&\quad\quad \textbf{for } i_3 = 1 \textbf{ to } n-1 \\
&\quad\quad\quad A[i_2, i_3] = 0.2 * (A[i_2, i_3] + A[i_2+1, i_3] \\
&\quad\quad\quad\quad\quad +A[i_2-1, i_3] + A[i_2, i_3+1] \\
&\quad\quad\quad\quad\quad +A[i_2, i_3-1]);
\end{aligned}
$$

By simply swapping the i_2 and i_3 loops, the performance will increase since the matrix now is accessed column-major instead of row-major, reducing the number of cache misses.

The KAP and SGI compilers simply unroll the inner-most loop instead of performing tiling transformations.

A threaded version of the basic SOR Nest is produced by taking the inner-most loop and substituting it with a thread:

$$
\begin{aligned}
&\textbf{for } i_1 = 1 \textbf{ to } t \\
&\quad \textbf{for } i_3 = 1 \textbf{ to } n-1 \\
&\quad\quad th\_fork(Compute, i_3, 0, A(0, i3-1), A(n, i3+1), 0); \\
&th\_run(0);
\end{aligned}
$$

$$
\begin{aligned}
&Compute(i_3, dummy): \\
&\quad \textbf{for } k = 1 \textbf{ to } n-1 \\
&\quad\quad A[k, i_3] = 0.2 * (A[k, i_3] + A[k+1, i_3] \\
&\quad\quad\quad +A[k-1, i_3] + A[k, i_3+1] + A[k, i_3-1]);
\end{aligned}
$$

Although there are data dependencies among threads, the algorithm works fine because the goal is to reach convergence. Since there is only one data structure used in each thread, only one dimensional scheduling is required. The number of threads created is $t*(n-1)$ Depending on the available cache size, the hints can be fine tuned to keep as much of the array as possible in the cache.

Table 6 shows the performance of three versions (untiled, hand tiled and threaded) of the SOR Nest with the size of the matrix $n = 2005$, the number of iterations of a single SOR step $t = 30$, and tile size $s = 18$. The KAP and SGI compilers perform only loop unrolling transformations instead of tiling, so we have included a hand tiled version [29] instead.

	R8000	R10000
Untiled	30.54	12.81
Hand Tiled	26.90	4.27
Threaded	23.10	4.31

Table 6: SOR performance in seconds.

On the R8000 (with 2MB L2 cache), the threaded version creates 60,120 threads in 63 bins with an average of 954 threads per bin. The threaded version reduces about 20% of the total running time of the untiled version on the R8000 system while it runs almost three time as fast as the untiled version on the R10000 system. The hand tiled version gives only a small improvement on the R8000, but it is slightly faster than the threaded version on the R10000.

R8000	Untiled	Hand-tiled	Threaded
I fetches	1,205,767	1,917,178	1,212,039
D references	482,042	703,522	483,973
L1 misses	90,451	5,259	90,631
rate	5.4	0.2	5.3
L2 misses	7,545	282	263
rate	3.6	0.2	0.1
L2 compulsory	251	268	258
L2 capacity	7,294	0	6
L2 conflict	0	13	0

Table 7: SOR memory references and cache misses in thousands.

Table 7 reports the simulated cache miss data for SOR on the R8000. Most L2 cache misses of the untiled version are capacity misses. Both hand-tiled and threaded versions can remove almost all capacity misses.

The hand-tiled version has 933M more references due to tiling the L1 cache, but it reduces the L1 cache misses by about 85M and reduces the L2 cache misses by about 7.3M. If we crudely assume that each instruction takes a single CPU cycle, the estimated overhead of 933M instructions would be about 12.4 seconds. By assuming that the overhead of an L1 and L2 cache miss is 7 cycles and 1.06 microseconds respectively, the estimated time saved by reducing L1 and L2 cache misses is 7.3 and 8.0 seconds respectively. Thus, the total estimated time saved is about 2.9 seconds, not so far from the actual time saved (3.6 seconds).

The threaded version has about 8.2M more references and 180K more L1 cache misses than the untiled version. By assuming that each instruction takes a cycle to execute and an L1 cache miss takes 7 cycles [23], the estimated overhead is about 0.1 second. The threaded version, on the other hand, reduces the number of L2 cache misses by 7.3M, most of which are capacity misses. The total estimated time saved is about 7.9 seconds, close to the actual time saved (7.4 seconds).

The crude analysis again does not take into many important details into account. It only offers some implications of where the time was saved from.

4.4 N–body

N–body is a program that uses the Barnes-Hut algorithm [6, 36] to solve a three dimensional N body problem. Each body has a position, mass, and velocity. Every body exerts a gravitational force on every other body. The program computes the motion of the bodies during some number of time steps.

For each time step (iteration) the algorithm creates a Barnes-Hut (BH) tree and inserts each body in it. Once the BH tree is created the program loops over a vector of all the bodies using the BH tree to calculate a new position for each body. The new position of all the bodies can be computed in parallel, i.e., there are no data dependencies between the bodies when the new position is being calculated. Creating the BH tree is quite fast, but calculating the new positions is slow. Our profiling shows that the program spends more than 88% of its time computing the new positions of the bodies.

Unlike the dense linear algebra programs above, N–body is a irregular and dynamic program. The data structures representing the bodies and the nodes of the tree are small. The positions of the bodies change at each iteration, and the BH tree is rebuilt for each iteration. Since no memory reference information available at compile time, automatic tiling is not feasible.

The threaded version computes the new positions by forking one thread per body with three hints: the x, y, and z coordinates of the body. We normalized the positions to the unit cube and then scaled them to the dimensions of the scheduling plane. Thus, threads in the same scheduling block were computing the new positions of bodies that near each other in space.

We ran both versions described above. On every iteration, the threaded version creates one thread for each body to calculate its new position, i.e., 64000 threads per iteration. Both programs computed the results for 64,000 bodies for four iterations.

	R8000	R10000
Unthreaded	153.81	53.22
Threaded	148.60	46.34

Table 8: N–body Performance in seconds.

Table 8 shows the execution times of both programs. The threaded version is faster than the unthreaded version on both machines. On the R10000, the threaded version is 15% faster. A typical iteration of the threaded version created 64000 threads in 46 bins with an average of 1,391 threads per bin. The distribution of threads per bin was much less uniform than in the other examples. This corresponds to the distribution of the bodies in the three dimensional space.

Table 9 shows the results of the cache simulations of

R8000	Unthreaded	Threaded
I fetches	1,820,656	1,838,089
D references	865,713	872,130
L1 misses	54,313	55,035
rate	2.0	2.0
L2 misses	1,674	778
rate	0.5	0.2
L2 compulsory	175	190
L2 capacity	1,131	495
L2 conflict	369	93

Table 9: N–body memory references and cache misses in thousands.

one iteration of the N–body codes for the R8000 system. Although the threaded version performs about 23.8M more instruction fetches and data references, there were only 722K more L1 misses. A crude estimate of the threading overhead is about 0.3 seconds for each N–body iteration.

Even though the unthreaded version of N–body has an L2 miss rate of only 0.5%, threading reduces it to 0.2%, decreasing the L2 capacity misses by a factor of 2.3. If we calculate the time saved in L2 misses using an L2 miss penalty of 1.06 microseconds, the estimated time saved is 0.95 seconds per iteration. Since the estimated overhead of threading is about 0.3 seconds, the estimated net time saving is about 0.65 seconds. The crude estimate differs from the actual time saved by about 0.75 seconds.

4.5 Sensitivity to Bin Dimension

The bin dimension is an important parameter in the scheduling algorithm. It determines the bin into which a thread is placed. The threaded version of the above experiments were all conducted with a fixed block size of which each dimension is $1/k$ of the second-level cache size for the k-dimensional case.

In order to understand how the block size affects the running time of our benchmarks, we ran them with varying sizes from 128K to 8M. We used some sizes larger than the second level cache size to confirm our intuitions. Figure 4 shows the results of these experiments on the Power Indigo2 with R8000 CPU. The X axis corresponds to the block dimension sizes and the Y axis corresponds to the running times.

These graphs show that when the sum of the block dimensions are less than or equal to the size of the second-level cache, the performance of the programs is relatively insensitive to the block size. For a program that is sensitive to L2 cache misses such as the matrix multiply program, there is a significant performance degradation when the block size is greater than the L2 cache size.

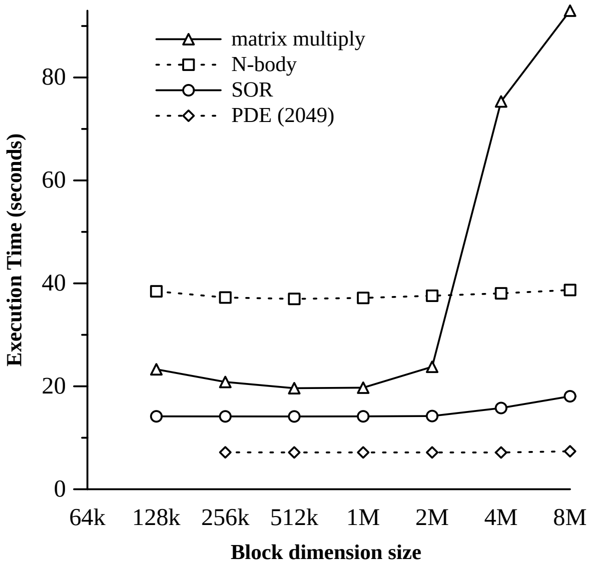

Figure 4: Execution times versus block sizes.

5 Related Work

There is a large body of related work including: arranging data structures to maximize locality of reference, tiling, multi-threaded architectures, and implementations of lightweight thread packages.

Early research efforts have proposed ways of rearranging data structures and altering algorithms to reduce page faulting in virtual memory [16, 1]. Tiling has become a well-known software technique for using the memory hierarchy effectively [18, 29]. Tiling can be applied to any levels of memory hierarchy, including virtual memory, caches, and registers. In particular, tiling can be very effective for scientific programs, but automatic tiling transformations are quite limited and manual transformation is difficult.

Scheduling on multiprocessors based on cache affinity is a technique that maintains processor/thread locality on multiprocessors [38]. Affinity scheduling avoids cache fills and invalidations on cache coherent multiprocessors when threads are allowed to migrate among processors.

Multi-threaded architectures have been proposed and implemented that support multi-threaded parallel programs [2, 3, 26, 28, 39]. They typically have architectural support for scheduling threads in one or a few CPU cycles in order to hide various latencies in pipelines, memory references, and network transactions. The success of this approach relies on creating threads efficiently at either compile time or run time.

Early ideas on Monitors provided the foundation for concurrent programming with threads in programming languages [22, 30]. The existence of multiprocessors have motivated several efforts to design and implement kernel and user-level thread packages [4, 7, 9, 13, 15] and very lightweight thread packages [25, 31]. However, none of these efforts have addressed the issue of scheduling threads for cache locality.

6 Limitations

The method described in this paper has several limitations. First of all, it attempts to reduce misses only for the largest cache (the cache connecting to the DRAM system). The overhead of the thread primitive operations is too high to schedule threads to reduce first level cache misses. On the other hand, when static memory reference information available, the tiling method used in compilers can take advantage of registers, first-level, and second-level caches.

Our experiments are limited to 3 address hints, though the scheduling algorithm can use k (where $k > 3$) addresses as hints to schedule threads. Applications in our experiments also do not need more than three addresses.

The thread package supports only independent, "run-to-completion" threads. Such an application programming interface can be conveniently used to program asynchronous iterative algorithms. However, it would not be convenient to program algorithms that have complex dependencies. Methods to specify dependencies and ways to implement them efficiently remain to be demonstrated.

Our experiments are also limited. We only experimented with four applications. We compared our results with only the tiled versions by the KAP compiler or the new SGI compiler. It would be more useful to compare our results with the results from a compiler with more advanced tiling techniques.

Our cache simulations for the R8000 system could only provide us with memory reference and cache miss information. The simulation does not take many important details into account. For example, it works with virtual addresses whereas the L2 cache uses physical addresses. It does not model the cache hierarchy in detail so that there is no information on CPU stalls due to the presures on caches. Crude analysis could only offer implications where saved time comes from. To figure out exactly where time goes, it would require detailed simulations of the R8000 CPU pipelines and its memory hierarchy.

Finally, we did not perform cache simulations for the R10000 and thus we are unable to analyze the performance results on that machine in detail.

7 Conclusions

This paper describes a method that uses fine-grained threads to improve the locality and execution time of programs. Our results with an experimental thread package show that this method can effectively reduce second-level cache misses and that the overhead of the thread package is small. In the best case (matrix multiply), the method can improve the performance of the untiled version by a factor of 2 on a system with R10000 and by a factor of 5 on a system with the R8000 CPU. In other cases, threads can also be used to improve performance by reducing L2 cache misses. We expect that latency tolerance techniques such as thread scheduling will become more important as the performance gap between memory and CPU increases.

Using a thread package as runtime support to reduce second-level cache misses can be useful for applications that have no information about memory reference patterns at compile time. This is demonstrated by both the PDE and N–body applications. On the other hand, when static information is available, a good compiler can do better than our thread package. This is because compiler tiling can reduce total memory references and L1 cache misses, as well as L2 cache misses.

The study in this paper represents an initial exploration of runtime thread scheduling to tolerate memory latency. Several questions are still open. For example, it is not clear whether the scheduling algorithm can be efficiently implemented with a general-purpose thread package that supports synchronization and preemptive scheduling. It appears that the idea proposed in this paper can be extended in a straightforward manner to improve performance on symmetric multiprocessors, but this remains to be demonstrated.

8 Acknowledgements

We would like to thank Thomas Anderson, Susan Eggers, Kathryn McKinley, and other programming committee members and reviewers for helpful comments and suggestions on the initial draft of the paper. Monica Lam, David Padua, and John Ruttenberg answered numerous questions about tiling and helped us find a good compiler for our experiments. Torsten Suel provided the N-Body simulation as well as useful comments on its data locality. Dave Parry, Bill Huffman, and Steve Schroder provided us with the cache information on the SGI machines. Michael Smith, Mike Uhler, and Marty Itzkowitz helped us to understand Pixie address trace formats.

Kai Li is supported in part by DARPA grant N00014-95-1-1144, and by NSF grants MIP-9420653 and CCR-9423123.

References

[1] W. Abu-Sufah, D.J. Kuck, and D.H. Lawrie. Automatic Program Transformations for Virtual Memory Computers. In *Proceedings of the 1979 National Computer Conference*, pages 969–974, June 1979.

[2] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of International Conference on Supercomputing*, pages 1–6, 1990.

[4] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[5] Thomas E. Anderson, Edward D. Lazowska, , and Henry M. Levy. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[6] J. E. Barnes and P. Hut. A hierarchical $O(NlogN)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[7] Brian Bershad, Edward Lazowska, Henry Levy, and David Wagner. An Open Environment for Building Parallel Programming Systems. In *Proc. ACM/SIGPLAN Conference*

on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS), pages 1–9, June 1988.

[8] Brian N. Bershad, J. Bradley Chen, Dennis Lee, and Theodore H. Romer. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of The 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[9] Andrew D. Birrell, John V. Guttag, Jim J. Horning, and Roy Levin. Synchronization Primitives for a Multiprocessor: A Formal Specification. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 94–102, November 1987.

[10] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, July 1995.

[11] Bernard Chazelle. Private communication, 1996.

[12] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, LaJolla, CA, June 1995.

[13] Thomas Doeppner Jr. Threads: A System for the Support of Concurrent Programming. Technical Report CS-87-11, Computer Science Department, Brown University, June 1987.

[14] C. C. Douglas. Caching in with multigrid algorithms: problems in two dimensions. Technical Report TR/PA/95/15, CERFACS, Toulouse, France, 1995.

[15] Richard P. Draves and Eric C. Cooper. C Threads. Technical Report CMU-CS-88-154, School of Computer Science, Carnegie-Mellon University, June 1988.

[16] J.L. Elshoff. Some Programming Techniques for Processing Multi-Dimentional Matrices in a Paging Environment. In *Proceedings of the NCC*, 1974.

[17] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Efficient Support for Fine-Grain Parallelism on Shared-Memory Machines. Technical Report TR 96-1, Department of Computer Science, The University of Arizona, January 1996.

[18] D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, October 1988.

[19] John L. Hennessy and Norman P. Jouppi. Computer Technology and Architecture: An Evolving Interaction. *IEEE Computer*, 24(9):18–29, September 1991.

[20] Mark D. Hill. DineroIII 3.4 Documentation (Unpublished Unix-style Man Page), June 1991.

[21] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1629, December 1989.

[22] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[23] Peter Y. Hsu. Design of the R8000 Microprocessor. Submitted to IEEE MICRO, 1993.

[24] Bill Huffman. Private communication, 1996.

[25] Suresh Jagannathan and James Philbin. A Customizable Substrate for Concurrent Languages. In *Proceedings of ACM SIGPLAN '92 Conference on Programming Languages Design and Implementation*, 1992.

[26] Stephen W. Keckler and William J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, 1992.

[27] R. Kessler and M.D. Hill. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Transactions on Computer Systems*, 10(4):338–359, November 1992.

[28] J. Kowalik, editor. *Parallel MIMD computation : the HEP supercomputer and its applications*. MIT Press, 1985.

[29] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of The 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[30] B. M. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[31] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[32] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX*, January 1996.

[33] *MIPS R10000 Microprocessor User's Manual*. MIPS Technologies, Inc, version 1.1 edition, 1996.

[34] Dave Parry. Private communication, 1996.

[35] Steve Schroder. Private communication, 1996.

[36] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *Computer*, 26(7):42–50, July 1993.

[37] Michael D. Smith. Tracing with Pixie. Technical Report CSL-TR-91-497, Computer Science Department, Stanford University, November 1991.

[38] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[39] Dean Tullsen, Susan Eggers, and Henry Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, June 1995.

[40] William A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.