Code and Data Transformations for Improving Shared Cache Performance on SMT Processors

Dimitrios S. Nikolopoulos

Department of Computer Science, The College of William & Mary,
McGlothlin–Street Hall, Williamsburg, VA 23187–8795, U.S.A.,
dsn@cs.wm.edu

Abstract. Simultaneous multithreaded processors use shared on-chip caches, which yield better cost-performance ratios. Sharing a cache between simultaneously executing threads causes excessive conflict misses. This paper proposes software solutions for dynamically partitioning the shared cache of an SMT processor, via the use of three methods originating in the optimizing compilers literature: dynamic tiling, copying and block data layouts. The paper presents an algorithm that combines these transformations and two runtime mechanisms to detect cache sharing between threads and react to it at runtime. The first mechanism uses minimal kernel extensions and the second mechanism uses information collected from the processor hardware counters. Our experimental results show that for regular, perfect loop nests, these transformations are very effective in coping with shared caches. When the caches are shared between threads from the same address space, performance is improved by 16–29% on average. Similar improvements are observed when the caches are shared between threads from different address spaces. To our knowledge, this is the first work to present an all-software approach for managing shared caches on SMT processors. It is also one of the first performance and program optimization studies conducted on a commercial SMT-based multiprocessor using Intel's hyperthreading technology.

Keywords: multithreaded processors, compilers, memory hierarchies, runtime systems, operating systems.

1 Introduction

Simultaneous Multithreaded Processors (SMT) were introduced to maximize the ILP of conventional superscalars via the use of simultaneous instruction issue from multiple threads of control [16]. SMT architectures introduce modest extensions to a superscalar processor core, while enabling significant performance improvements for both parallel shared-memory programs and server workloads. To improve cost-effectiveness, threads on SMT processors share all processor resources, except from the registers that store the private architectural state of each thread. In particular, SMT processors use shared caches at all levels. From a performance perspective, a shared cache used by a multithreaded program accelerates inter-thread communication and synchronization, because threads

A. Veidenbaum et al. (Eds.): ISHPC 2003, LNCS 2858, pp. 54–69, 2003.
© Springer-Verlag Berlin Heidelberg 2003

that share an address space can prefetch data in the cache for each other [8]. However, for scientific programs which are organized to fully utilize the cache, a shared cache organization introduces significant problems. The working sets of different threads that share a cache may interfere with each other causing an excessive amount of conflict misses. Unless the cache space of the processor is somehow partitioned and "privatized", the problem of coping with thread conflicts on shared caches can become a serious performance limitation.

In this paper, we present software solutions for partitioning shared caches on SMT processors, based on standard program transformations and additional support from the OS, or information collected from the processor hardware counters. Our solutions work in scientific codes with perfect loop nests, which are tiled for improved temporal locality. Tiling is perhaps the most important locality optimization in codes that operate on dense multidimensional arrays. It is used extensively in scientific libraries such as LAPACK. It is also the target of numerous compiler optimizations for memory hierarchies [3, 4, 6, 9, 12, 17].

Our solutions combine dynamic tiling with either copying, or modification of the array layouts in memory. Dynamic tiling amounts to changing the tile size of the program at runtime, if a program senses that its threads are competing for shared cache space with other threads. Competition can be self-interfering (i.e. multiple threads of the same address space compete for cache space on the same SMT), or cross-interfering (i.e. threads from different address spaces compete for shared cache space on the same SMT). We opt for dynamic tiling instead of static adjustment of tile sizes for a shared cache, to give the program an opportunity to adapt its tile size without a-priori knowledge of the assignment of processors and intra-processor threads to the program. The program can adapt its execution to configurations that use one thread per processor or multiple threads per processor, as well as to multiprogrammed execution with threads from multiple address spaces sharing a processor.

Copying is a method for reducing the intra-tile and inter-tile interference in the cache [15]. Copying has been used along with other compiler optimizations for reducing conflict misses. We use copying as a means to confine the tiles used by a thread to a fixed set of cache locations. Coupled with dynamic adjustment of the tile size, copying achieves an implicit spatial partitioning of the cache between the working sets of competing threads to improve performance.

For codes that do not lend themselves to copying due to either correctness or performance implications, we use block layouts of arrays in memory and interleaved assignment of blocks to processors, to achieve a similarly effective spatial partitioning of the cache between tiles belonging to different threads. This solution is expected to be more widely applicable than copying. However, it must cope with restrictions posed by the programming language on the layout of arrays in memory.

The proposed methods have some technical implications. They require a mechanism to detect cache interference between threads at runtime, as well as mechanisms to control the tile size and allocate memory blocks for copying tiles to. We provide solutions to these problems, using either additional support

from the operating system or information collected from the processor hardware counters.

To the best of our knowledge, this paper is the first to propose the use of standard code and data layout transformations to improve the management of shared caches on SMT processors without modifications to the hardware. It is also among the first to measure the impact of the proposed transformations on an actual commercial multiprocessor with SMT processors. We use two simple kernels, a tiled parallel matrix multiplication and a tiled point SOR stencil to illustrate the effectiveness of the proposed optimizations. We present experiments from a four-processor Dell PowerEdge 6650 with Xeon MP processors. The processors use Intel's hyperthreading technology and allow simultaneous execution of instructions from two threads. We illustrate the benefits of dynamic tiling, copying and block data layouts using both stand-alone executions of the programs with multiple threads per processor and multiprogrammed workloads with threads from different programs sharing processors.

The rest of this paper is organized as follows. Section 2 details the proposed transformations. Section 3 presents techniques to detect cache contention due to sharing between threads at runtime. Section 4 presents our experimental setting and results. Section 5 provides an abbreviated review of related work. Section 6 concludes the paper.

2 Dynamic Tiling, Copying and Block Data Layout

Figure 1 illustrates the use of the cache if two threads that run disjoint parts of a tiled loop nest share the cache, assuming a perfectly tiled code with a tile size that fits exactly in the cache. Each thread individually tries to use the entire cache space C, by selecting the appropriate tile size. Since the cache can only fit one tile of size $\leq C$, there are excessive conflict misses, while each thread is trying to reuse data in the tile on which it is currently working. These conflict misses lead to poor cache utilization and poor performance.

The immediate solution that comes to mind is to reduce the tile size to a fraction of the cache, e.g. $C/2$ in the case of two threads sharing the cache. The problem with this solution, besides a potential reduction of spatial locality due to smaller tile sizes [4], is that it does not reduce cache conflicts, as shown in Figure 2. Despite the smaller tile size, each thread tries to load two tiles in the cache. Even if each thread individually avoids self-interference between its own tiles, there are still conflicts between tiles belonging to different threads.

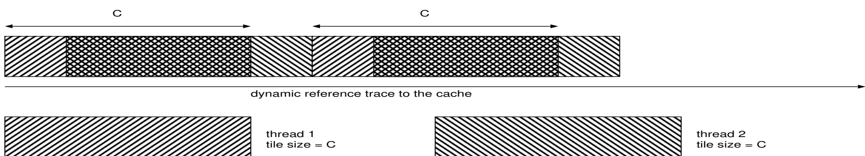

Fig. 1. Cache use when full-size tiles from two threads are loaded in the cache.

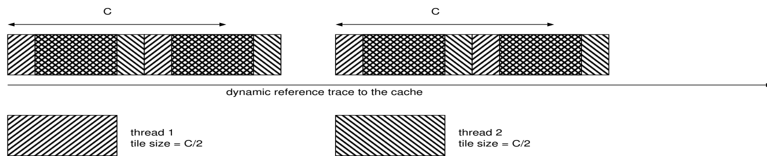

Fig. 2. Cache use when half-size tiles from two threads are loaded in the cache.

```
for (jj = 0; jj < n; jj += tj)
    for (kk = 0; kk < n; kk += tk)
        for (k = kk; k < MIN(n,kk+tk); k++) {
            for (j = jj; j < MIN(n,jj+tj); j++)
                b_block[(k-kk)*tj+(j-jj)] = b[k*n+j];
            for (i = 0; i < n; i++)
                for (k = kk; k < MIN(n,kk+tk); k++)
                    for (j = jj; j < MIN(n,jj+tj); j++)
                        c[i*n+j] += a[i*n+k]*b_block[(k-kk)*tj+(j-jj)];}
```

Fig. 3. A tiled matrix multiplication with copy.

One way to solve this problem is to use smaller tiles and at the same time ensure that tiles from the same thread occupy always the same set of cache blocks. A simple method to accomplish this is to use tile copying [15]. Each thread allocates a memory block equal to the tile size. During the execution of the loop nest, the thread copies the working tile into the memory block, executes the loop nest iterations that work on the tile and then copies the tile back to the array, if the tile is modified during the iterations. Figure 3 shows the tiled matrix multiplication with copy.

Copying fixes the virtual address range and the set of cache locations occupied by a tile, while the tile is being reused in the loop nest. If the thread selects a tile size which is a fraction of C, copying guarantees that during the loop, all tiles use mostly the specified fraction of the cache. Care must still be taken so that tiles from different threads do not interfere in the cache. To address this problem we consider two cases, tiles belonging to different threads in the same address space and tiles belonging to threads in different address spaces.

For tiles belonging to the same address space, the problem can be solved with proper memory allocation of the buffers to which tiles are copied. In practical cases, virtual addresses are mapped to cache lines modulo the number of sets (for set-associative caches) or the number of cache lines (for direct-mapped caches). Assume that the cache needs to be partitioned between the tiles of T threads. We select a tile size which is the best size for a cache of size C/T and apply copying. If we assume that the program uses $N = PT$ threads to execute the loop nest, P the number of multithreaded processors, we allocate the buffers to which tiles are copied in virtual addresses $r_0, r_0 + \frac{C}{T}, \ldots, r_0 + (N-1)\frac{C}{T}$, where r_0 is a starting address aligned to the cache size boundary. After memory allocation is performed, the runtime system schedules threads so that two threads the tiles

of which conflict in the cache are never scheduled on the same processor. More formally, if two tiles with starting addresses $r_0 + i\frac{C}{T}, r_0 + j\frac{C}{T}, 0 \leq i < j \leq N-1$, are used by two different threads, the threads should be scheduled on the same SMT processor only if $mod(j,i) \neq 0$. For the case of 2 threads per processor, this means that threads that work on even-numbered tiles should be scheduled on even-numbered hardware contexts and threads that work on odd-numbered tiles should be scheduled on odd-numbered hardware contexts. In simpler words, tiles should be interleaved between hardware contexts. In the general case of multithreaded processors with T threads per processor, two threads using tiles with starting addresses $r_0 + i\frac{C}{T}, r_0 + j\frac{C}{T}$ should not be scheduled on the same processor if $mod(i,T) = mod(j,T)$.

For tiles belonging to different address spaces, guaranteeing conflict avoidance is difficult because a thread running tiled code does not have any knowledge of how a competing thread uses the shared cache. Copying can only guarantee that each thread that runs tiled code reuses a fixed set of locations in the cache. If all the threads that share the cache run tiled computations, the conflict problem will be alleviated with dynamic tiling and copying if the cache is set-associative, which is the case in all modern microprocessors. Otherwise, if we assume that each program can use only local information to control its caching strategy, the problem is intractable.

If threads need to change their tile sizes at runtime, they need to allocate a new memory block for each tile size used. The additional memory consumption is therefore equal to the sum of the different tile sizes that a thread uses at runtime, times the number of threads in the program. For the transformation considered here, each thread uses at most two tile sizes and the space overhead is $PT(C + \frac{C}{T})$. The number of memory allocations is equal to the number of different tile sizes used at runtime, times the number of threads in the program i.e. $2PT$. Since memory allocations of buffers can be executed in parallel, the actual overhead is 2 memory allocations.

Copying can not be used for free. It incurs instruction overhead and additional cache misses while data is copied from arrays to buffers. We use the analysis of [15] to balance the earnings with the overhead of copying, assuming that all accesses to the reused tile will conflict with probability $1/T$. We estimate the number of cache accesses and cache misses during innermost loop nest iterations that reuse the tile, as well as the number of accesses and instructions executed while copying a tile from an array to a buffer. Then, we compare the cost of conflict misses to the cost of copying, by associating a fixed latency for each cache miss and each copy instruction. The average latency of cache misses and instructions is obtained by microbenchmarking the target processors. We use PAPI [1] for this purpose.

Unfortunately, copying may still not be applicable because of its runtime cost. This is frequently the case when both copy-in and copy-out operations must be used to maintain data consistency. An alternative solution to copying is to use array layouts that achieve the same effect as copying on the cache. Block data layouts [10] can be used for this purpose. With a block data layout, an array

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

Fig. 4. The array on the right shows the block data layout of the 4×4 array on the left, using 2×2 blocks.

is first decomposed in blocks, which correspond to the tiles used when tiling is applied in the loop nest that accesses the array. The elements of each block are stored in contiguous locations in memory as shown in Figure 4. Previous work has shown that a block data layout can be equally or more efficient than a conventional data layout due to good use of the memory hierarchy and the TLB [10]. In our context, the block data layout has a second hidden advantage. It lets us control explicitly the location of the virtual address space in which each block is stored.

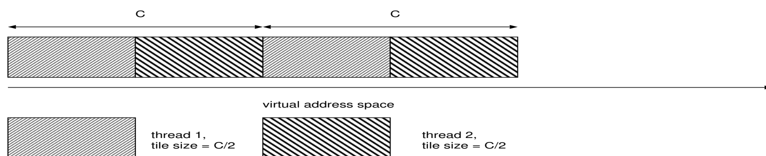

Fig. 5. Block data layout with interleaved assignment of blocks to threads.

If copying can not be used, we use the block data layout and position the blocks in the virtual address space so that blocks used by the same thread in the parallel execution are placed with a distance of C between each other, C being the cache size. As shown in Figure 5, blocks belonging to different threads are interleaved in memory and regions of the address space with size equal to the cache size C are partitioned between blocks. Conceptually, we try to force conflicts between blocks of the same thread and avoid conflicts between blocks of different threads. For caches shared between threads from different programs, interference may still occur, but this solution ensures that each thread will only compete for an equal share of the cache, instead of all cache locations.

To implement dynamic tiling and copying we create two versions of the tiled part of the loop nest. The first version executes the properly tiled loop, where the tile size is selected to utilize the entire cache space. The second version executes the loop nest with the reduced tile size and tile copy. More formally, if the loop nest has n levels, $L_1 \ldots L_n$, out of which $m < n$ levels are the outermost loop control levels and $n{-}m{+}1$ are the innermost tiled levels, the first version executes the tiled levels $L_m \ldots L_n$, whereas the second version executes $n - m + k, k > 1$ levels, including the tiled levels $L_m \ldots L_n$ using a small tile size that utilizes $\frac{1}{T}$ of

the available cache space and $k-1$ outermost levels used to partition the original tiles into smaller tiles. The selection between the two versions is done with a conditional that checks if cache contention between multiple threads occurs at runtime. The implementation of this conditional is discussed in more detail in Section 3. Switching between the two versions is done at level L_m in the original loop nest, i.e. at tile boundaries. This simplifies the code significantly but may delay the adaptation to cache contention by as much as the time required for the processing of one tile. The buffer space used for copying is statically preallocated in the code.

If the block data layout is used, the array is transformed from the standard to the block layout and two versions of the loop nest are created again. Both versions use the block data layout. In this case, the base tile size is selected to utilize $\frac{1}{T}$ of the available cache space. In the first version, each thread accesses combined large tiles, each composed of T consecutive smaller tiles at level L_m. In the second version, each thread accesses the smaller tiles with a stride equal to T, so that tiles are interleaved between threads. The first version of the code executes the $n-m$ innermost tiled loops using the compound tile, whereas the second version executes the $n-m$ innermost tiled loops of the original code using the small tile size and one more loop level to control the interleaving of tiles between processors. Again, a conditional is used to select between the two versions.

We experimented with several compile-time algorithms for selecting the tile size and we ended up using the one proposed by Chame and Moon [3]. Chame and Moon's algorithm computes an initial set of tile sizes that avoids self-interference. It calculates a set of rectangular tile sizes starting with a tile of one row with size equal to the cache size and incrementally increasing the number of rows by one, while setting the maximum row size equal to the minimum distance between the starting addresses of any two tile rows in the cache. For each tile size of this set, the algorithm performs a binary search on all possible tile row sizes, to find the tile row size that minimizes capacity misses and cross-interference misses. The binary search is based on the assumption that capacity misses decrease monotonically with an increasing row size and cross-interference misses increase monotonically with an increasing row size. The estimated capacity misses are calculated analytically for each array reference from the parameters of the loop nest. The cross-interference misses are estimated probabilistically from the cache footprints of array references in each tile.

After extensive experimentation we found that Chame and Moon's algorithm tends to perform consistently better than other heuristics [3, 4, 12] when the tile size is reduced to cope with cache sharing. We note however that it is beyond the scope of this paper to investigate thoroughly the relative performance of tiling algorithms on SMT processors.

A point worth noting is that we dynamically change the tile size to the best tile size for a cache with size equal to $\frac{1}{T}$ of the actual cache size. This implies that each thread is expected to use as much cache space as possible, which is valid for tiled parallel codes. Nevertheless, in arbitrary workloads (for example, with

multiprogramming, or if the application uses one thread for communication and one thread for computation), situations may arise in which one thread requires little or no cache space, whereas another thread running on the same processor requires all the available cache space. We leave this issue open for investigation in future work.

A drawback of using block data layouts is that if for any reason the standard language layouts (row-major or column-major) of the arrays must be preserved, additional copying must be used to create the block layout before the loop nest and restore the original layout after the loop nest. In the codes we used for the experiments presented in this paper, both of which are composed of a single loop nest, creating the block layout before the loop nest and restoring the original layout after the loop nest did not introduce significant overhead. We measured the overhead for data set sizes between 90 and 200 Megabytes and the arithmetic mean was 1.1% of the total execution time.

We have implemented the aforementioned transformations by hand in two regular tiled codes, a blocked matrix multiplication and a 2-D point SOR kernel. Work for formalizing and automating these transformations in a compiler is in progress. Since we are interested in the performance of parallel code, it is necessary to partition the tiles between processors. We use a simple blocked partitioning scheme in which each processor is assigned a stack of contiguous tiles. We used OpenMP and the default static scheduling algorithm for parallelization. This simple scheduling scheme leads often to load imbalance, but this effect is mitigated when the number of tiles per thread is increased.

3 Detecting Cache Contention

If the compiler could know in advance the number of threads used by a program and the processors on which these threads are going to execute, it could selectively generate code with or without the transformations that optimize execution for shared caches. Unfortunately, the number of threads is usually a runtime parameter and the processors on which threads execute is unknown until runtime. In most shared-memory multiprocessors, where programmers use either POSIX threads or OpenMP to parallelize, the number of threads is set by either an environment variable or a call to a library function. For a P-processor machine with T-way multithreaded processors, and a program that uses $2 \leq t < PT$ threads, the operating system may decide to co-schedule threads on the same processor(s) or not, depending on the load and the scheduling policy.

Furthermore, although a single program running on an idle machine is most likely going to execute with a fixed number of threads pinned to processors and the OS can avoid cache sharing simply by not co-scheduling multiple threads of the same program on the same processor, if the machine runs multiple programs and/or system utilities at the same time, the scheduler may decide to change the number and the location of threads dynamically. Last but not least, a single parallel program may wish to utilize all processors and threads within processors. In this case, cache sharing is inevitable. In general, it is desirable for a program to

identify locally at runtime whether there is cache sharing or not. We examine two mechanisms to accomplish this goal, one that uses an extension to the operating system and one that uses the processor hardware counters.

The first solution is to extend the operating system with an interface that exports the placement of threads on processors to a page in shared memory, which is readable by user programs. Earlier work has coined the term shared arena to characterize this interface [5]. A shared arena can be used by a user-level threads library in many ways for improving thread scheduling and synchronization algorithms under multiprogrammed execution. In our case, we use the shared arena to export a bit vector that shows which hardware contexts are busy running either user-level code, or operating system handlers. The OS sets a bit whenever it assigns a thread or an interrupt handler to run on a hardware context. We used Linux, in which the hardware contexts are treated as virtual CPUs, with multiple virtual CPUs per physical processor. Each virtual CPU is seen as a separate processor by the software. The tiled code that we generate checks this vector each time it picks a new tile to work on. The problem with an OS extension, even a very simple one, is that it is not portable and requires kernel modifications which may or may not be possible. We investigated a second option, which is to try to implicitly detect cache sharing by measuring the cache misses with hardware performance counters while the program is executing. Hardware performance counters have become a standard feature of modern microprocessors, therefore a solution based on them would be significantly more portable. Using hardware performance counters, we compare the actual number of cache misses per tile, against the expected number of cache misses, which is calculated analytically, using stack distances [2].If the measured number of cache misses exceeds the expected number by a threshold, we assume interference due to cache sharing. The threshold is allowed to compensate for potential self and cross-interference between data of the same thread. We have experimented with several threshold values and found that a threshold equal to the expected number of cache misses is always an accurate indication of contention for cache space.

4 Experimental Setting and Results

We experimented on a 4-processor Dell PowerEdge 6650 Server. Each processor is a 1.4 GHz Xeon MP with hyperthreading technology. The processor can execute simultaneously instructions from two threads, which appear to the software as two different processors. We call these processors virtual processors onwards, to differentiate from the actual physical processors, which are multithreaded. Each physical processor has an 8 KB L1 data cache integrated with a 12 KB execution trace cache, an 8-way associative 256 KB L2 cache and an integrated 512 KB L3 cache. The system runs Linux 2.4.20, patched with a kernel module that binds threads[1] to virtual processors. The processor binding module is not a scheduler. We use it only to specify the placement of threads on virtual processors and

[1] In Linux, threads and processes are synonyms, each thread has a unique PID, threads can share an address space and thread switching is performed in the kernel.

enforce cache sharing, in order to run controlled experiments. Linux provides no other way to achieve the same effect at user level. Nevertheless, we note that the dynamic tiling algorithm is scheduler-independent. It is designed to detect cache contention and sharing, regardless of the processor allocation policy of the OS.

We used two codes, a tiled matrix multiplication and the 5-point stencil loop of a 2-D point SOR code, both parallelized with OpenMP. Copying was used in the matrix multiplication, whereas block data layout with proper block alignment was used in the SOR code. Copying is affordable in matrix multiplication but not in SOR. The dynamic tiling transformation was applied for the L1, the L2 and the L3 cache recursively. We used large data sets, so that the performance of the codes becomes more bound to the L2 and L3 cache latencies. Note that the caches maintain the inclusion property. The programs were compiled with the Intel C compiler, version 7.1, using the O3 optimization level. Besides standard code optimizations, the compiler performs profile-guided interprocedural optimization and function inlining, partial redundancy elimination, automatic vectorization of inner loops and automatic padding of arrays for improved cache performance. Padding was enabled in matrix multiplication but disabled in SOR to control explicitly the data layout. Cache contention was detected by measuring L1 cache misses using PAPI.

Figure 6 shows the execution time of the tiled matrix multiplication using fixed tile sizes and dynamic tiling with copy. The matrix multiplication is executed as a stand-alone program on an otherwise idle machine. We run the code with 1 thread on one processor, 2 threads on 2 physical processors (labelled 2×1), 4 threads on two physical processors (labelled 2×2), 4 threads on 4 physical processors (labelled 4×1) and 8 threads on 4 physical processors (labelled 4×2).

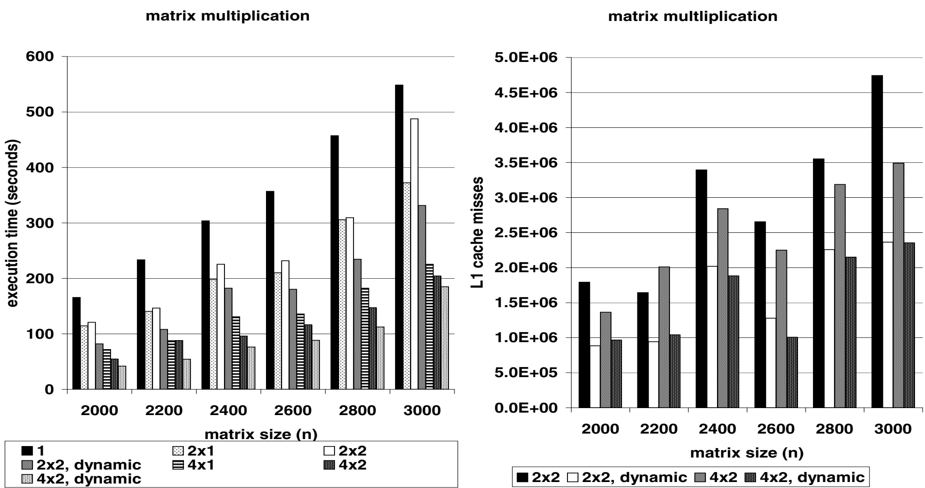

Fig. 6. Execution times (left chart) and number of L1 cache misses (right chart) of an $n \times n, 2000 \leq n \leq 3000$, matrix multiplication with fixed tile sizes and dynamic tiling with copying.

The label suffix *dynamic* indicates the use of tiling with dynamic reduction of the tile size when cache contention is detected, instead of fixed tile sizes that use the entire cache space. The execution time savings for two threads running on two processors average 26% (standard deviation is 5.29) and for eight threads running on four processors 15.66% (standard deviation is 8.53). The average speedup (measured as the arithmetic mean of the parallel execution time for all matrix sizes, divided by the execution time with 1 thread), is improved from 1.41 to 1.91 on 2 SMTs running 4 threads and from 2.95 to 3.89 on 4 SMTs running 8 threads. The sublinear speedups are attributed to load imbalance due to uneven distribution of tiles between threads and memory hierarchy effects, including limitations of bus bandwidth and bus contention.

It is interesting to notice that simultaneous multithreading on a single processor does not always improve performance. With fixed tile sizes, using 4 threads on 2 processors leads to a performance drop of 9.3% (arithmetic mean across all matrix sizes), compared to the performance achieved with 2 threads running on 2 processors. On the contrary, with dynamic tiling, performance is improved by 17.95% on average in the same setting. With 8 threads on 4 processors, performance is always improved, compared to the performance achieved with 4 threads on 4 processors. The arithmetic mean of the improvement is 15.44% with fixed tile sizes and 35.44% with dynamic tiling.

The right chart in Figure 6 shows the number of L1 cache misses throughout the execution of the programs with fixed tile sizes and dynamic tiling, using 4 threads on 2 SMTs and 8 threads on 4 SMTs. Each bar is the arithmetic mean of the number of cache misses of all threads, during the execution of the matrix multiplication loop nest, excluding initialization.On two processors with two threads each, L1 cache misses are reduced by 54.63% on average (standard deviation is 6.34). On four processors with two threads each, L1 cache misses are reduced by 61.15% on average (standard deviation is 10.57). L2 cache misses (not shown in the charts) are reduced by 51.78% and 55.11% respectively. Although cache misses are reduced more with 8 threads than with 4 threads, the parallel execution time is not improved similarly.

Figure 7 shows the execution times and L1 cache misses of the tiled SOR code. The observed results are very similar to matrix multiplication and show that the block data layout transformation is almost equally effective. Execution time improvements average 19.45% with 4 threads (standard deviation is 11.32) and 29.19% with 8 threads (standard deviation is 13.24). Speedup is improved from 1.76 to 2.17 with 4 threads on 2 SMTs and from 3.28 to 4.65 with 8 threads on 4 SMTs. Using 2 threads instead of 1 thread on the same SMT leads to marginal performance losses or improvements with fixed tile sizes (in the range of $\pm$ 5%) and more significant improvements (ranging from 12.23% to 32.42%) with dynamic tiling. L1 cache misses are reduced by 50.55% on average (standard deviation is 10.86). L2 cache misses (not shown in the charts) are reduced by 59.44% on average (standard deviation is 16.12).

Figure 8 shows the average execution time of two tiled matrix multiplications running simultaneously with four threads each, using a stride-2 allocation of

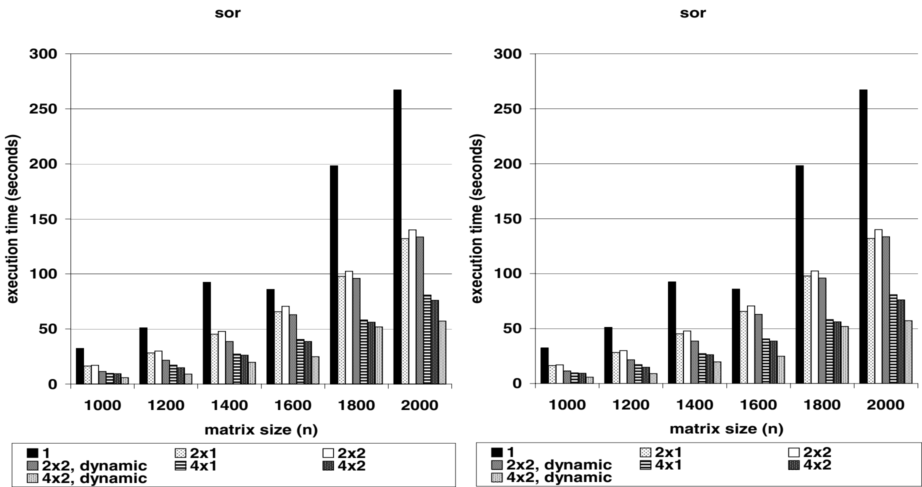

Fig. 7. Execution times (left) and number of L1 cache misses (right) of an $n \times n, 1000 \leq n \leq 2000$, point SOR code tiled in two dimensions, with fixed tile sizes and dynamic tiling with block data layout.

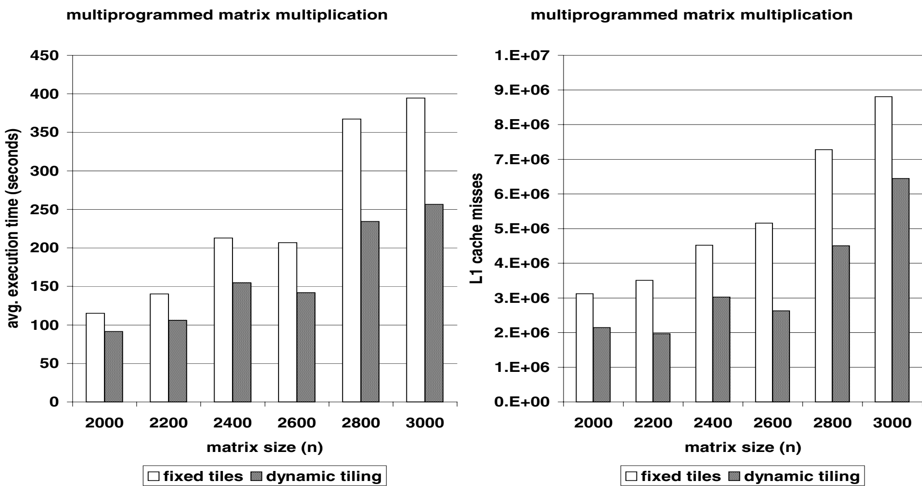

Fig. 8. Average execution time and number of L1 cache misses of two tiled matrix multiplications executed with four threads each, using stride-2, odd-even allocation of virtual processors to threads.

virtual processors to threads. This means that one matrix multiplication runs on even-numbered virtual processors and another matrix multiplication runs on odd-numbered virtual processors simultaneously. This experiment evaluates if dynamic tiling and copying can alleviate cache contention between independent tiled codes sharing the cache. The arithmetic mean of the reduction of execution

time is 28.9% (standard deviation is 5.83). L1 cache misses are reduced by 61.24% on average. L2 cache misses (not shown) are reduced by 71.79% on average.

In the specific experiment, cache contention is detected by the hardware performance counters. We repeated the experiment and used sharing notifications from the OS instead of the processor hardware counters. We observed measurable but not significant improvements. We ran one more experiment to test whether our techniques work equally well with a more dynamic workload competing for cache space with the transformed programs. We executed the tiled matrix multiplication with four threads, using a stride-2 processor allocation. Concurrently, we launched a script with a blend of integer, floating point, system call, disk I/O and networking microbenchmarks, adapted from the Caldera AIM benchmark (version s9110). The script was modified so that individual microbenchmarks were distributed between the odd-numbered virtual processors, while the even-numbered virtual processors were executing one parallel matrix multiplication. The script creates significant load imbalance between the processors, as some of the benchmarks (e.g. I/O and networking) make intensive use of caches due to buffering, while others (e.g. system calls) make little or no use of caches.

Figure 9 shows the execution time and L1 cache misses of matrix multiplication with this workload, for various matrix sizes. Performance is improved by 24.69% on average (standard deviation is 4.47). L1 cache misses are reduced by 64.35% on average (standard deviation is 9.89). During the experiments, we instrumented the code to check how long the code was executing with the reduced tile size and how long it was executing with the larger tile size. A smaller tile size was used during 94.53% of the execution time. The code switched twice

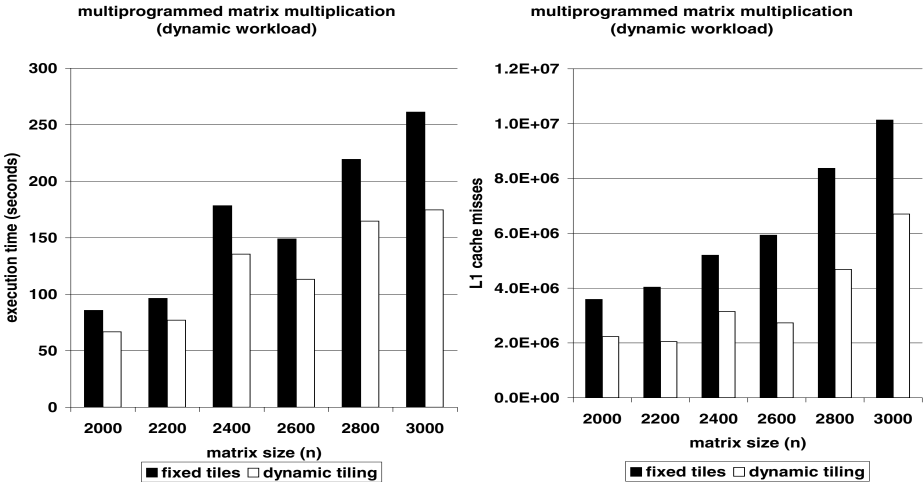

Fig. 9. Execution time and L1 cache misses of one tiled matrix multiplication with four threads running on even-numbered virtual processors, together with the multiprogramming workload of Caldera's AIM benchmark, spread among the four odd-numbered virtual processors.

from bigger to smaller tiles and vice versa. Some further testing has shown that theoretically, the adaptive code should increase more frequently the tile size and reduce execution time even further, because the idle intervals, as indicated by the load of each processor, account for more than 15% of the execution time of the tiled code. On the other hand, when cache misses were used as the sole indication of contention, the adaptation of the code was correct and the proper tile size was used in all cases. We plan to examine this issue in more detail in future work.

5 Related Work

Simultaneous multithreaded processors have been a focal topic in computer systems research for almost 8 years, initially along the hardware design space [16] and later, along the operating system design space [8, 11]. From the hardware design perspective, all studies concluded that shared caches at all levels of the cache hierarchy yield the most cost-effective design. From the software perspective, the related work has focused on parallel programming issues such as synchronization and memory allocation, but not on program optimizations for shared caches.

Tiling and multilevel blocking of array codes for sequential and parallel machines has been investigated in an enormous number of publications, only a sample of which is referenced here [3, 4, 6, 7, 9, 12, 15, 17, 18]. Recently, researchers have turned their attention to managing space-shared caches on multiprogrammed systems, via a number of hardware and software techniques, including static cache partitioning [13] and memory-aware job scheduling [14]. These studies place the burden of effective management of shared caches on the operating system. Otherwise, they propose hardware cache partitioning. We have presented a software method based on standard compiler transformations, which requires little or no additional support from the operating system.

6 Conclusions and Future Work

This paper presented a set of program transformations, namely dynamic tiling, copying and block data layouts, which alleviate the problem of conflicts between threads on the shared caches of simultaneous multithreaded processors. These transformations result to significant performance improvements, both for standalone multithreaded programs and for programs competing for shared cache space in multiprogrammed execution environments. The strength of the contribution of this paper is the ability to use standard and well-known code and data layout transformations to cope with a problem (thread interference in shared caches) which has not been addressed and has been merely investigated outside the hardware context. Given that both multithreaded processors and chip multiprocessors with shared caches are gaining more ground in the market, investigating program optimizations in the software context will contribute to the development of better compilers and other system software tools for these systems.

The weaknesses of the contribution of this paper parallel the weaknesses of practically all transparent locality optimization frameworks. They are effective in limited cases, mostly numerical scientific codes with "easy to analyze" loop nests and data access patterns. One direction of future work is the investigation of the development of more generic cache-conscious program and data layout schemes for processors with shared on-chip caches. We are also seeking a convenient analytical framework for quantifying conflict misses in shared caches, to derive automatic program transformations. One more unexplored issue which arises from this work is the use of dynamic instead of static cache partitioning methods, based on the cache footprints of threads.

Acknowledgments

Xavier Martorell implemented the kernel module for binding threads to virtual processors, which was used in this paper after some minor modifications. Christos Antonopoulos implemented C macros for interfacing with PAPI and solved several technical problems of PAPI on the Xeon MP processors. This research was supported by a startup research grant from the College of William&Mary and NSF Research Infrastructure Grant EIA-9972853.

References

1. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proc. of Supercomputing'2000: High Performance Networking and Computing Conference*, Dallas, TX, Nov. 2000.
2. C. Cascaval and D. Padua. Estimating Cache Misses and Locality using Stack Distances. In *Proc. of the 17th ACM International Conference on Supercomputing (ICS'2003)*, pages 150–159, San Francisco, CA, June 2003.
3. J. Chame and S. Moon. A Tile Selection Algorithm for Data Locality and Cache Interference. In *Proc. of the 13th ACM International Conference on Supercomputing (ICS'99)*, pages 492–499, Rhodes, Greece, June 1999.
4. S. Coleman and K. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proc. of the 1995 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'95)*, pages 279–290, San Diego, CA, June 1995.
5. D. Craig. An Integrated Kernel and User-Level Paradigm for Efficient Multiprogramming. Technical Report CSRD No. 1533, University of Illinois at Urbana-Champaign, June 1999.
6. I. Kodukula, N. Ahmed, and K. Pingali. Data-Centric Multilevel Blocking. In *Proc. of the 1997 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'97)*, pages 346–357, Las Vegas, Nevada, June 1997.
7. N. Mateev, N. Ahmed, and K. Pingali. Tiling Imperfect Loop Nests. In *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, TX, Nov. 2000.

8. L. McDowell, S. Eggers, and S. Gribble. Improving Server Software Support for Simultaneous Multithreaded Processors. In *Proc. of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'2003)*, San Diego, CA, June 2003.
9. K. McKinley, S. Carr, and C. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.
10. N. Park, B. Hong, and V. Prasanna. Analysis of Memory Hierarchy Performance of Block Data Layout. In *Proc. of the 2002 International Conference on Parallel Processing (ICPP'2002)*, pages 35–42, Vancouver, Canada, Aug. 2002.
11. J. Redstone, S. Eggers, and H. Levy. Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IX)*, Cambridge, MA, Nov. 2000.
12. G. Rivera and C. Tseng. A Comparison of Tiling Algorithms. In *Proc. of the 8th International Conference on Compiler Construction (CC'99)*, pages 168–182, Amsterdam, The Netherlands, Mar. 1999.
13. G. Suh, S. Devadas, and L. Rudolph. Analytical Cache Models with Applications to Cache Partitioning. In *Proc. of the 15th ACM International Conference on Supercomputing (ICS'01)*, pages 1–12, Sorrento, Italy, June 2001.
14. G. Suh, L. Rudolph, and S. Devadas. Effects of Memory Performance on Parallel Job Scheduling. In *Proc. of the 8th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'02)*, pages 116–132, Edinburgh, Scotland, June 2002.
15. O. Temam, E. Granston, and W. Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing when Data Copying Should be Used to Eliminate Cache Conflicts. In *Proc. of the ACM/IEEE Supercomputing'93: High Performance Networking and Computing Conference (SC'93)*, pages 410–419, Portland, OR, Nov. 1993.
16. D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95)*, pages 392–403, St. Margherita Ligure, Italy, June 1995.
17. M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *Proc. of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Toronto, Canada, June 1991.
18. J. Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, Aug. 2000.