

---

## Dynamic tiling for effective use of shared caches on multithreaded processors

---

Dimitrios S. Nikolopoulos

Department of Computer Science,  
College of William and Mary, Williamsburg, VA, USA  
E-mail: dsn@cs.wm.edu

**Abstract:** Simultaneous multithreaded (SMT) processors use data caches which are dynamically shared between threads. Depending on the processor workload, sharing the data cache may harm performance due to excessive cache conflicts. A way to overcome this problem is to physically partition the cache between threads. Unfortunately, partitioning the cache requires additional hardware and may lead to lower utilisation of the cache in certain workloads. It is therefore important to consider software mechanisms to implicitly partition the cache between threads by controlling the locations in the cache in which each thread can load data. This paper proposes standard program transformations for partitioning the shared data caches of SMT processors, if and only if there are conflicts between threads in the shared cache at runtime. We propose transformations based on dynamic tiling. The key idea is to use two tile sizes in the program, one for single-threaded execution mode and one suitable for multithreaded execution mode and switch between tile sizes at runtime. Our transformations combine dynamic tiling with either copying or storing arrays in block layout. The paper presents an implementation of these transformations along with runtime mechanisms for detecting cache contention between threads and react to it on-the-fly. Our experimental results show that for regular, perfect loop nests, these transformations provide substantial performance improvements.

**Keywords:** multithreaded processors; compilers; memory hierarchies; runtime systems; operating systems.

**Reference** to this paper should be made as follows: Nikolopoulos, D.S. (XXXX), 'Dynamic tiling for effective use of shared caches on multithreaded processors,' *Int. J. High Performance Computing and Networking*, Vol. X, Nos. Y, pp.000–000.

**Biographical notes:** Dimitrios S. Nikolopoulos is an Assistant Professor of Computer Science at the College of William and Mary. Before joining William and Mary, he was a Visiting Research Assistant Professor with the Coordinated Science Laboratory at the University of Illinois, Urbana-Champaign. He received his Diploma Degree in Computer Engineering in 1996 and his PhD in Computer Engineering in 2000, both from the University of Patras in Greece. His research focuses on the development of programming environments for effective and effortless adaptation of software to high-end computing platforms. He has authored more than 40 papers on runtime optimisation techniques and operating system support for high-performance and high-throughput computing. He is a recipient of an NSF CAREER Award and four 'Best Paper Awards' at SC'2000, IPDPS'2002, CCGrid'2002 and ISHPC-V.

---

## 1 Introduction

Simultaneous multithreaded (SMT) processors were introduced to maximise the ILP of conventional superscalars via the use of simultaneous instruction issue from multiple threads of control (Tullsen et al., 1995). SMT architectures introduce modest extensions to a superscalar processor core, while enabling significant performance improvements for both parallel programs and multiprogram workloads.

SMT architectures partition most of the processor resources between simultaneously executing threads. Some of the resources are statically partitioned while others are dynamically partitioned. Statically partitioned resources are managed by dividing evenly their available capacity between the active threads. Static partitioning isolates

threads from contention. However, it often leads to low utilisation of resources, if there are threads which cannot fully utilise their shares. Dynamically partitioned resources are allocated to threads on demand. Dynamic sharing enables higher resource utilisation at the cost of higher hardware complexity (Tullsen et al., 1995). Intermediate schemes that combine static sharing with dynamic switching from multithreaded to single-threaded execution mode and vice versa have also been proposed as cost-effective design points for SMTs.

As an example, the Intel Xeon MP processor, a version of Pentium IV based largely on the original SMT design (Cross 2002), uses statically shared load and store buffers and a statically shared reorder buffer. In contrast, the data cache of the processor is dynamically shared at all levels.

From a performance perspective, a shared data cache used by a multithreaded program accelerates interthread communication and synchronisation (McDowell et al., 2003). Unfortunately, sharing the data cache may also introduce excessive cache conflicts. Many programs, particularly scientific applications, depend heavily on the memory hierarchy and the ability of the program to use effectively the data cache.

Tiling of iteration spaces and array blocking (Wolf and Lam, 1991) are among the most popular and most extensively studied automatic program optimisation techniques for improving data locality and cache performance in scientific codes. The essence of tiling and blocking is to organise data and computation so that each processor accesses data in blocks that fit in the cache and each block is accessed for as many innermost loop iterations as possible, so that the processor has a chance to satisfy most memory accesses from the cache instead of main memory. In practically all cases, tiling and blocking are applied so that the working set of the program (or a thread therein) uses the entire cache space (Anderson et al., 1995; Carr and Lehoucq, 1997; Chame and Moon, 1999; Coleman and McKinley, 1995; Kodukula et al., 1997; McKinley et al., 1996; Rivera and Tseng, 1999). Unfortunately, if such transformations are applied directly to a SMT processor with a shared data cache, the working set of the threads that share the cache may interfere with each other. Unless the cache space of the processor is ‘privatised’ between the simultaneously executing threads, performance will suffer due to cache conflicts.

In this paper, we present standard program transformations for implicit partitioning of the shared data caches of SMT processors. The transformations are based on well-known techniques for blocking iteration and data spaces. They require additional support from the OS or information collected from the processor hardware counters to detect if cache contention between threads occurs at runtime. Our solutions work in scientific codes with perfect loop nests, which are tiled to improve temporal locality. The proposed transformations benefit any processor that uses a shared data cache, including multithreaded processors and chip multiprocessors.

We propose the use of dynamic tiling, that is, generating code which switches on-the-fly between two tile sizes. One tile size is selected to use the entire cache space, while another is selected to use a fraction of the cache space, which is inversely proportional to the number of threads sharing the cache. The selection of the tile size depends on whether the cache is actively shared between threads or not at runtime. To fix the cache locations that each thread uses at runtime and avoid conflicts, we combine dynamic tiling with either block copy or a conversion of the default array layouts to block layouts. In both cases, the result is that the blocks loaded by a thread on the shared cache are always confined to the same set of cache locations, and two threads may store their blocks in the cache simultaneously. We opt for dynamic tiling instead of static adjustment of the tile size to adapt tiled codes to multiprocessor SMTs. On multiprocessor SMTs, programs or the operating system may use one thread or multiple threads per processor depending on several workload parameters. As the number of threads that will be used to execute a tiled code and the placement of these threads

on the SMT processors are not known a priori, it is preferable to generate tiled code which can adapt to both single-threaded and multithreaded execution on each SMT at runtime.

We use three simple kernels, a tiled parallel matrix–matrix multiplication, a tiled point SOR stencil and a tiled LU decomposition, to illustrate the benefits of the proposed optimisations. We present experiments from a multiprocessor Dell PowerEdge 6650 with Xeon MP processors. The Xeon MP uses Intel’s hyperthreading technology, a design based largely on the original SMT concept. We illustrate the advantages of dynamic tiling, copy and block data layouts using both standalone parallel benchmarks and multiprogram workloads.

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

## 2 Dynamic tiling for shared data caches

Figure 1 shows a simplified version of a tiled matrix–matrix multiplication ( $m \times m$ ), a classic example of tiling a loop nest for temporal locality. The code is tiled so that array  $b$  is accessed in blocks of size  $tk \times tj$ . The block dimensions  $tk$  and  $tj$  are selected so that a block of  $b$ , a row of  $c$  and a column of  $a$ , which cumulatively form the working set of the loop nest, fit simultaneously in the cache. The original loop nest is expanded by two levels, which walk over the blocks of  $b$ . We use  $m \times m$  as a working example in the following discussions.

Figure 2 illustrates what would be the ideal use of a data cache from a properly tiled  $m \times m$  running on a single thread and assuming that the working set of the tiled code fits perfectly in the cache. This implies that there is no self-interference (i.e. conflicts in the cache between elements of the same array in the same block) or cross-interference (i.e. conflicts in the cache between elements of different arrays in the same tile.)

Figure 3 illustrates the use of the data cache if two threads that execute two consecutive tiles of  $m \times m$  share the cache. Note that although the example assumes an interleaved assignment of tiles to threads, the same problem occurs with a block assignment. Each thread individually tries to use the entire cache space by selecting a correspondingly large tile size. As the cache can only fit a block of  $b$ , a row of  $c$  and a column of  $a$ , there are excessive self-interference misses between the working sets of the two tiles. In fact, if the two tiles are executed in perfect synchrony, their working sets will overlap completely in the cache, increasing the cost of conflicts up to the cost of a complete cache reload.

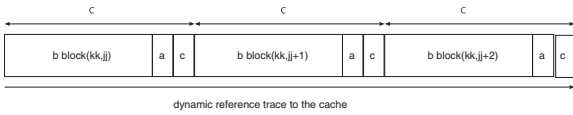
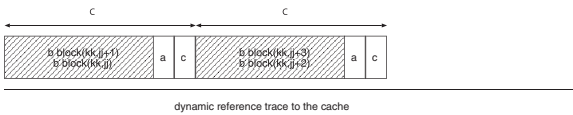
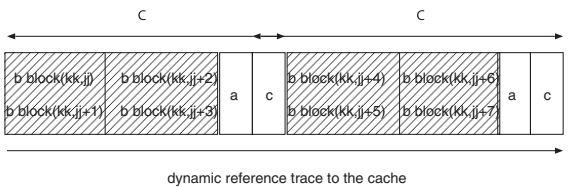
A first solution that comes to mind is to reduce the tile size to use a fraction of the cache size equal to  $1/T$ , where  $T$  is the number of threads sharing the cache. This implies that each thread will use a smaller tile size, hence a smaller block size in its working set. The problem with this solution, is that it may not reduce cache conflicts, as shown in Figure 4. Despite the smaller block size, each thread may try to load two

**Figure 1** A tiled  $m \times m$ 

```

for (jj = 0; jj < n; jj += tj)
  for (kk = 0; kk < n; kk += tk)
    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[k*n+j];
    
```

blocks of its own in non-overlapping cache locations. Even if each thread avoids self-interference between its own two blocks, there may still be conflicts between blocks belonging to different threads.

**Figure 2** Ideal use of a data cache from a properly tiled  $m \times m$ . The example assumes that the data cache is used by a single thread. In the ideal case, there is no self-interference or cross-interference between array elements inside a tile.  $C$  is the data cache size

**Figure 3** Cache conflicts when two threads share the cache and use tile sizes that utilize the entire cache space. Shaded areas indicate conflicts. Although the specific example assumes an interleaved (cyclic) assignment of tiles to threads, the same problem arises with a block assignment

**Figure 4** Cache conflicts when two threads that share the cache use only half of the available cache space for their blocks. Despite the smaller block size, each thread may try to load two (smaller) non-overlapping blocks of  $b$  in the cache. Even if each thread avoids self-interference between its own blocks, conflicts with the blocks in the working set of the other thread may still arise


## 2.1 Using copy

The previous discussion gives two hints for solving the problem of sharing the cache between threads running tiled computations. First, the cache needs to be partitioned between the threads to avoid conflicts. In fact, if the cache is not partitioned, the conflicts will overwhelm the gains from

making the entire cache available to each thread, precisely because each thread is structured to load and use the entire cache space. Previous studies of the performance of SMTs (Tuck and Tullsen, 2003) have indicated that when the SMT runs threads that execute identical code on different data from the same program, the number of cache misses explodes and harms performance.

Second, since the cache is not partitioned by the hardware, we need to not only reduce the tile size, but also provide a software mechanism to confine the working set of each tile to a fixed set of cache locations. If we guarantee that the cache locations allocated to a thread do not overlap with the cache locations allocated to other threads that share the cache, we can avoid conflicts.

A simple method to achieve the aforementioned effects in software is to copy the blocks used by the tiles of a thread from the original array to a buffer which is pinned to a fixed location in the virtual address space (Temam et al., 1993). Before the execution of a tile, the thread copies the temporarily reused block to the buffer. It then executes the tile and discards the block, if the block is read-only, or copies the modified elements in the block back to the array. Figure 5 shows a tiled version of  $m \times m$  that uses copy.

By using copy, we fix the range of virtual addresses and the set of cache locations used by the block in each tile. If the thread uses a block size which is a fraction of the size of the cache, copy guarantees that during the execution of the loop nest, all tiles executed by a single thread will be using mostly the same set of locations in the cache. Additional care must be taken so that blocks belonging to different threads do not interfere in the cache.

Interference between blocks can be avoided with proper memory allocation of the buffers used by different threads. Assume that the cache needs to be partitioned between the working sets of  $T$  threads. We select a tile size which is the best size for a cache of size  $C/T$  and then apply copy. If 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  $r_0, r_0 + C/T, \dots, r_0 + (N - 1)C/T$ , where  $r_0$  is an 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  $r_0 + i(C/T), r_0 + j(C/T)$ ,  $0 \leq i < j \leq N - 1$ , are assigned to two threads, these threads should be scheduled on the same SMT processor only if  $\text{mod}(j, i) \neq 0$ . In the case of two threads per processor, this means that threads that work on even-numbered tiles should be scheduled on even-numbered hardware threads and threads that work on odd-numbered tiles should be scheduled on

**Figure 5** A tiled  $m \times m$  with copy

```

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)];
  }

```

odd-numbered hardware threads.<sup>1</sup> In simple words, tiles should be interleaved between hardware threads. In the general case of multithreaded processors with  $T$  threads per processor, two threads using tiles with starting addresses  $r_0 + i(C/T)$ ,  $r_0 + j(C/T)$  should not be scheduled on the same processor if  $\text{mod}(i, T) = \text{mod}(j, T)$ .

The discussion so far assumes that it is always the case that tiled codes use multiple threads on an SMT processor. Multithreaded processors have hardware support to switch from multithreaded to single-threaded execution mode and commit all their resources to the execution of a single thread. On the Xeon MP processor, for example, one of the two threads running on the processor may be suspended using the `halt` instruction. The other thread is then free to use up all the resources of the processor, including the caches. If the processor executes in single-threaded mode, tiled codes need not shrink their tiles to enable cache partitioning.

Our assumption is that tiled codes may execute in either single-threaded or multithreaded mode, without a priori knowledge of mode switches that happen at runtime. A mode switch may happen either because the user chooses to do so, or because the compiler or the runtime system finds it better to execute a program in one of the two modes. For example, OpenMP, a popular standard for parallel programming on SMPs, allows the runtime system to control dynamically the number of threads used in a program. Thread control can be used either to enable adaptation of the program to multiprogram execution, or to coarsen the granularity of threads so that the program wastes less time in thread synchronisation. If the number of threads used is less than the number of processors times the number of hardware execution contexts per processor, the operating system has a choice between coupling threads on the same processor, or spreading threads across processors. The tiled code should work efficiently in both cases, that is, utilise the entire cache space in single-threaded mode and use a fraction of the cache space in multithreaded mode.

To enable adaptation to single-threaded and multithreaded execution at runtime, we allow tiled codes to use two tile sizes and choose between these sizes at runtime, using a conditional that evaluates whether the cache is shared or not between threads. The adaptive version of matrix multiplication is shown in Figure 6. We discuss the implementation of the conditional in Section 3. As the example shows, we pre-calculate the tile sizes for single-threaded and multithreaded execution mode and control the tile size at runtime on a tile by tile basis. Each tile is executed to adapt to either single-threaded or multithreaded mode, based on whether contention has been detected during the

execution of the previous tile of the same thread. If a mode switch happens, the tile size changes to adapt to the new execution conditions. Though not extensively optimised, this implementation is both simple and efficient.

Block copy consumes additional space for buffers. As each thread uses at most two tile sizes, hence two buffers for copying, the memory consumption is equal to twice the number of threads in the program. For the transformation proposed here, each thread uses at most two tile sizes and the space overhead is  $PT(C + (C/T))$ . The number of memory allocations is equal to the number of tile sizes used at runtime, times the number of threads in the program, that is,  $2PT$ . If the memory allocations for the buffers can be executed in parallel, the actual runtime overhead is equal to the cost of two memory allocations.

There is always a performance penalty associated with copy in tiled loop nests. There are both empirical and analytical heuristics (Temam et al., 1993; Wolf and Lam, 1991) to assess whether copy should be used or not at compile time. Instead of using one of these compile-time heuristics, we use the following simplified set of rules: First, we apply copy only if the blocks used in the tiled code are read-only. Our experience suggests that using copy-in and copy-out operations for read-write blocks in tiled loop nests is rarely beneficial. Second, we always apply copy if we detect cache contention between threads, based on the empirical observation that the cache reloads that threads must suffer due to conflicts are significantly more expensive than the copy overhead, particularly when the latter is amortised across many tile iterations that reuse the same block.

## 2.2 Using block data layouts

To overcome the limitations of using block copy for solving the problem of conflicts on shared caches, we consider a second, more broadly applicable solution based on block array layouts (Park et al., 2002).

With a block layout, an array is decomposed into blocks, which correspond one-to-one to the blocks accessed by the tiles of an optimised loop nest. The elements of each block are stored in contiguous memory locations as shown in Figure 7. Assume for simplicity a two-dimensional array of size  $n \times n$  which is initially stored in row-major order. The linearised address of element  $(i, j)$  is  $i \times n + j$ . In the block layout, the new linearised address of element  $(i, j)$  can be calculated in a two-step process. Assume that the array needs to be decomposed into  $B \times B$  blocks of size

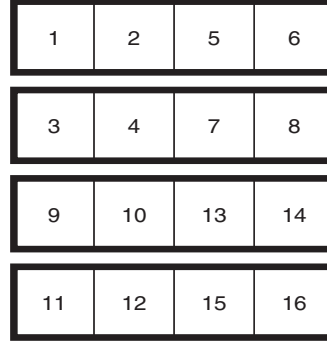
**Figure 6** An adaptive version of tiled  $m \times m$  with copy

```

calculate_tile_size(n,C,&ttj,&ttk);
calculate_tile_size(n,C/2,&tjj,&tkk);
tj = ttj; tk = ttk;
for (jj = 0; jj < n; jj += tj)
    for (kk = 0; kk < n; kk += tk) {
        if (cache_contention_present()) {
            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)];
            tj = tjj; tk = tkk; }
        else {
            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[k*n+j];}
            tj = ttj; tk = ttk; }}
    
```

**Figure 7** The right part of the figure shows the block layout of the  $4 \times 4$  array on the left, using blocks of size  $2 \times 2$ 

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



$b \times b$ . First, the block row and block column to which  $(i, j)$  belongs are calculated as

$$br = \left\lceil \frac{i}{B} \right\rceil \quad (1)$$

$$bc = \left\lceil \frac{j}{B} \right\rceil \quad (2)$$

The linear block number to which  $(i, j)$  belongs to is thus

$$br \times B + bc \quad (3)$$

The offset of element  $(i, j)$  within its block is

$$bo = (i \% b) \times b + (j \% b) \quad (4)$$

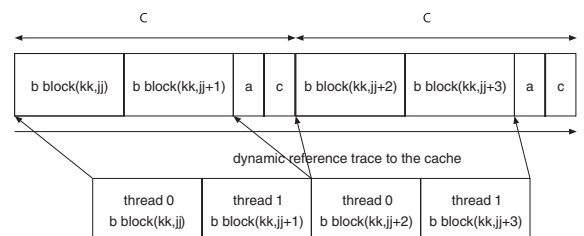
The linear location of element  $(i, j)$  in the new layout is given by

$$(br \times B + bc)b^2 + bo \quad (5)$$

Previous work has shown that block array layouts are often more efficient to access than conventional row-major or column-major layouts, due to the exploitation of temporal and spatial locality in the memory hierarchy (Park et al., 2002). In our context, the block layouts

can alleviate the problem of cache contention because the compiler can control the location in the virtual address space in which each block is stored in conjunction with the assignment of blocks to threads.

Figure 8 illustrates the main idea. The block size is adjusted as in the case of copy to let all threads that share the cache load a block in the cache simultaneously. Once the layout of the arrays is converted to block layout, the compiler can position the blocks in the virtual address space so that blocks used by the same thread are placed at a distance of  $C$  from each other,  $C$  the cache size. Consecutive blocks in memory are assigned to threads in an interleaved manner.

**Figure 8** Use of the shared data cache in  $m \times m$  when the block layout is used for  $b$ . Conflicts are avoided when blocks are properly aligned in memory and assigned to threads in an interleaved manner


**Figure 9** A tiled  $m \times m$  using block layout for array  $b$ . The complex expressions for mapping the original indices of  $b$  to the indices in the new layout can be eliminated if one observes that all accesses to  $b$  in the two innermost loops have the same  $block$  number and the offset of each element of  $b$  inside a block is a linear expression

```

for (jj = 0; jj < n; jj += tj)
  for (kk = 0; kk < n; kk += tk) {
    block = (jj/tj)*(n/tj) + (kk/tk);
    for (k = kk; k < MIN(n, kk+tk); k++)
      for (i = 0; i < n; i++) {
        mod = 0;
        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*tj*tk+mod];
            mod++;
          }
      }
  }

```

Two issues pertaining to the use of block array layouts need further investigation. The first is the translation of the array indices used with the standard layout in the transformed program that uses the block layout. As shown in equations (1)–(5), calculating the new location of each element requires expensive integer division and modulo operations. This problem can be overcome if one observes that in the innermost levels of the loop all accesses to the blocked array belong to the same block. The ID of this block can be calculated using the two outermost loop indices. Furthermore, within the same block, the offset of each element is a linear expression. Figure 9 gives an example of how the code can maintain locally the block number and offset without using expensive division operations. Note that the illustrated optimisation is not always applicable without further transformations. In the simplified example of  $m \times m$ , we assume that only  $b$  is in block layout and that the references to  $b$  inside the loop nest fall always within a block’s boundaries. The latter is not true in many cases, for example, in stencil codes where the accesses to the nearest neighbours of an element may cross block boundaries. To deal with these cases, we first peel iterations off the innermost loop nest so that all elements along the block boundaries are accessed with their indices as specified in equations (1)–(5) and optimise the accesses of all other elements inside the block. Alternatively, one may consider using extended block sizes and overlapping blocks that include the nearest neighbour block elements.

The second issue to resolve is the adaptation of the block size. We create two versions of the tiled loop nest, as in the case of copy. However, we do not maintain two copies of the blocked array, using the two different block sizes that correspond to single-threaded and multithreaded mode. Instead, one copy of the array with the smaller block size (i.e. the one used in multithreaded execution) is maintained. The loop nest is transformed to a two-version nest with a conditional. One version walks over the blocks of the array in chunks of  $T$ ,  $T$  the number of threads sharing the data cache, while the other walks the blocks one by one. This code generation scheme assumes that the block layout for the single-threaded mode subsumes the block layout for the multithreaded mode, in the sense that the size of the larger blocks used in single-threaded mode must be a multiple of the size of the smaller blocks used in

multithreaded mode. This strategy resembles hierarchical tiling (Carter et al., 1995).

### 2.3 Selecting the tile/block size

We experimented with several compile-time algorithms for selecting the tile size and we ended up using the one proposed by Chame and Moon (1999). 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, 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 in this set, the algorithm performs a binary search on all possible tile row sizes, to find the tile row size that minimises 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 when dynamic tiling is used, Chame and Moon’s algorithm tends to perform consistently better than other popular heuristics found in Coleman and McKinley (1995), Rivera and Tseng (1999). In addition, we observed that the tiles found by the algorithm yielded slightly better performance than fixed small tile sizes (e.g.  $24 \times 24$  or  $32 \times 32$ ) which are commonly chosen for L1 caches. We note nevertheless that it is beyond the scope of this paper to do a thorough investigation of the relative performance of existing algorithms for selecting tile sizes on SMT processors.

For the block layouts, we used the same algorithm and derived block sizes from tile sizes, assuming that the arrays used in the tiled code are dense. We have not experimented yet with sparse arrays.

### 2.4 Other remarks

When dynamic tiling is used and contention in the cache is detected at runtime, we set the tile size to the best size for a

cache with size equal to  $1/T$  of the 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 (e.g. with multiprogram execution, or when 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. This issue is not addressed in this work. If a dynamically tiled code runs in a multiprogram workload, we expect that dynamic tiling will always reduce to some extent the induced conflict misses, regardless of the behaviour of the competing programs on the processor. We verify this intuition with some experiments presented in Section 4.

A drawback of using blocked data layouts is that if for any reason the standard language layouts (row major or column major) of the arrays must be preserved across function invocations, extra copy operations 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 because the execution time of the computation was relatively large compared to the overhead of copy. However, with small data sets, one should consider the granularity of work inside the loop nest before applying the block data layout, so that the transformation pays off in performance.

We have implemented the aforementioned transformations by hand in three regular tiled codes, a blocked  $m \times m$ , a 2D point SOR kernel and a blocked LU factorisation. Work for formalising and automating these transformations in a compiler tool based on Open64 is in progress. As we are interested in the performance of parallel code, we parallelised the kernels with OpenMP. To partition the tiles between processors, we used an interleaved (cyclic) assignment of tiles to processors, which is consistent with the requirements of block data layouts.

### 3 Detecting cache contention

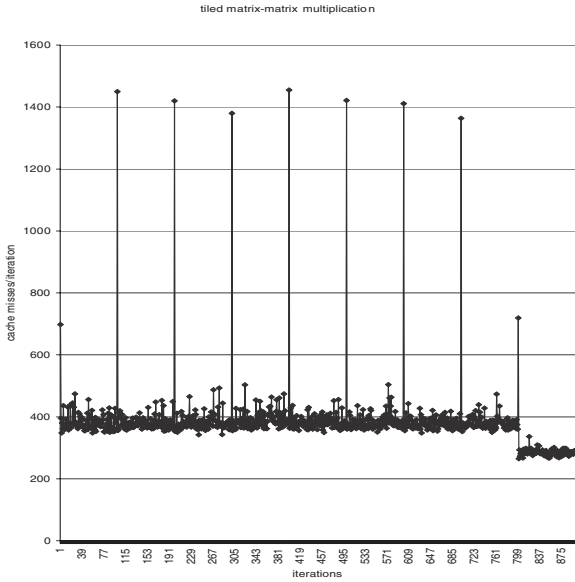
To detect cache contention at runtime so that tiled codes can adapt to both single-threaded and multithreaded execution on an SMT processor, we consider two mechanisms. The first mechanism is a direct mechanism that uses either information provided from the operating system, or a user-level utility like `top` or the `/proc` filesystem, to identify if a thread on a processor is idling or executing in a process address space. These mechanisms were very easy to implement in Linux. For all scheduling purposes, Linux treats the hardware threads on the same processor as equally accessible CPUs, each with a unique CPU id. The kernel maintains information about the process currently running on each thread in the `/proc` filesystem. When the system is in idle mode, hardware threads are used for the `init` process, the kernel event daemon (`keventd`) and the soft IRQ daemon (`ksoftirqd_CPUPX`,  $X = 0, 1, \dots$ ).

If a user-level process is fired up on any of the CPUs, the event is reflected immediately in the `/proc` filesystem. We implemented both a kernel module that accesses the `/proc` filesystem directly and a user-level module based on `top` to collect the state of all CPUs. This state is communicated to user programs as a bit vector. When a pair of consecutive bits starting at an even-numbered location in the vector is set, two threads on the same CPU are sharing the CPU resources. Once sharing is detected, the program infers that there is contention for accessing the cache. Obviously, we do not account the kernel idle loop and event daemons as sources of cache contention.

Using a kernel module or `top` is arguably a non-portable solution. It relies on Linux and its current interface. 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 are now found in all modern microprocessors. Tools such as PAPI (Browne et al., 2000) provide simple and portable interfaces for accessing hardware counters in a machine-independent manner.

We use the following simple rule for detecting cache contention using hardware counters. Assume that the tiled loop nest is composed of  $L$  levels, out of which  $L_c$  is the number of the outermost tile control levels (the ones that walk the blocked array tile by tile),  $L_r$  is the number of reuse levels (i.e. the set of iterations in the loop nest across which one block is reused) and  $L_b$  the innermost levels within a tile. Let  $1 \leq i_1 \leq u_1, \dots, 1 \leq i_{L_c} \leq u_{L_c}$  be the indices and iteration spaces of the loops at the reuse levels. Without loss of generality assume that each loop nest is normalised to begin at iteration 1. Starting from iteration 1,  $1, \dots, 2$ , and for every iteration until iteration  $u_1, \dots, u_{L_c} - 1$ , we use the hardware counters to calculate the number of L1 cache misses per iteration. Note that all these iterations reuse the same tile, so the number of cache misses is expected to be reduced from the number of cache misses during iteration  $1, \dots, -1$  by at least as many cache misses as required to load the tile. If during one of the iterations  $1, \dots, 2$  through  $u_1, \dots, u_{L_c} - 1$  we observe a number of cache misses which are ‘close’ to the number of cache misses we observed during iteration  $1, \dots, 1$ , we infer contention and we activate the multithreaded version of the tiled code. We set a threshold (e.g. 90% of the number of cache misses of reuse iteration  $1, \dots, 1$ ) and trigger the contention flag once this threshold is exceeded.

Figure 10, illustrates the number of cache misses per reuse iteration for the tiled  $m \times m$  of Figure 1, using matrices of size  $100 \times 100$ . The reuse level of the loop is the third level (indexed by  $i$ ) and each block of  $b$  is reused for 100 iterations. It is easy to observe the spikes in the number of cache misses that happen every 100 iterations, when a new block of  $b$  is loaded in the cache. During intervals of 100 iterations of the reuse loop ( $i$ ), the number of cache misses stays mostly flat. Most spikes raise at around 1400 misses. Using a 90% threshold if one iteration  $i, i = 1, \dots, 99$  incurs more than 1260 misses, we trigger the contention flag and trigger the change of the tile size in the code.

**Figure 10** Cache misses per reuse iteration for a tiled  $m \times m$ 

In practice, the aforementioned heuristic works very well, compared also to analytical heuristics that we used in earlier stages of this work. Therefore, we recommend this heuristic as a simple method for detecting contention at runtime.

#### 4 Experimental setting and results

We experimented on a four-processor Dell PowerEdge 6650 server. Each processor is a 1.4 GHz Xeon MP using hyperthreading technology. The processor can execute simultaneously instructions from two threads, which appear to the operating system as two different processors and can run either a single parallel program or a multiprogram workload. We use the term *hardware threads* for the hardware execution contexts within a processor, to differentiate from the term *processor*, which implies a multithreaded CPU. Each processor has an 8 KB L1 data cache integrated with a 12 KB execution trace cache, an eight-way associative 256 KB L2 cache and an integrated 512 KB L3 cache. The L1 data cache and the L2 and L3 caches are dynamically shared between the two threads. The system runs Linux 2.4.20, patched with a kernel module that pins kernel threads<sup>2</sup> to hardware threads. This module is not a scheduler in itself. We use it only to control the experiments by pinning kernel threads to run on the same processor and share the cache, if needed. This is necessary because the native Linux scheduler does not necessarily run two kernel threads on the same processor, if a program uses less kernel threads than the total number of hardware threads in the system. For example, with our module we can run an experiment with four kernel threads running on two processors, whereas with Linux the four threads may either run on two processors, or be spread across three or four processors of the machine, depending on an instantaneous decision of the kernel scheduler.

We used three codes, the tiled  $m \times m$  shown earlier, a 5-point stencil of a 2D point SOR code (shown in Figure 11), and a tiled LU decomposition (shown in Figure 12), all parallelised at the outermost loop level with

OpenMP. In the  $m \times m$  we used both the copy transformation and the block array layouts for all the arrays in the code. In the SOR kernel and the LU decomposition we used only the block array layout because the overhead of copy exceeds the gain from reducing conflicts. The reason behind this behaviour in SOR is that both copy-in and copy-out operations need to be used and there is not enough reuse of each block of  $a$  within a tile. In LU, copy-in and copy-out operations also offset the gains from copying. Tiling as well as the transformation for changing dynamically the tile size were applied for the L1, the L2 and the L3 cache recursively. We note that the second and third levels of tiling did not add much to the performance of the codes, a result which was also observed in Rivera and Tseng (1999). We used relatively large data sets, so that the performance of the codes becomes more bound to the L2 and L3 cache miss latencies. Note that the caches maintain the inclusion property.

The programs were compiled with the Intel C compiler, Version 7.1, at the maximum optimisation level. Besides standard code optimisations, the compiler performs profile-guided interprocedural optimisation and function inlining, partial redundancy elimination, automatic vectorisation of inner loops and automatic padding of arrays for improved cache performance. Padding was disabled when block layouts were used, so that the compiler does not change the layout of the transformed arrays. The compiler was successful in vectorising the assignment statements in the innermost iterations of  $m \times m$  only. We do not include vectorisation in the results presented here, as it was not uniformly applicable in the experiments. We plan to investigate the integration of vectorisation with OpenMP parallelisation in tiled codes in the near future. In these experiments, cache contention was detected dynamically, by measuring the L1 cache misses per reuse iteration using PAPI (see Section 3).

Figure 13 shows the execution time of the tiled  $m \times m$  using a fixed tile size selected by Chame and Moon's algorithm, dynamic tiling with copy, and dynamic tiling with block data layout for  $b$ , as shown in Figure 9. The  $m \times m$  was executed alone on an otherwise idle machine. We ran the code with one thread on one processor, two threads on two processors (labelled  $2 \times 1$ ), four threads on two processors (labelled  $2 \times 2$ ), four threads on four processors (labelled  $4 \times 1$ ) and eight threads on four processors (labelled  $4 \times 2$ ). The label suffix *copy* indicates the use of dynamic tiling with copy, while the label suffix *block layout* indicates the use of block array layouts. The results are grouped according to the size of the arrays in the experiment.

The first thing to notice on each set of bars, is that if there is no sharing of the data cache between threads on the same processor, there is little or no overhead incurred from the extra code that runs for detecting cache contention, both when the code uses copy and when the code uses the block layout. This can be observed by comparing in each set of 11 bars for a given matrix size, the second, third and fourth bars (two threads running on two processors, with no cache sharing) and the eighth, ninth and 10th bars (four threads running on four processors, with no cache sharing). The overhead ranges from 0.2% to 1.5% over the execution time of the tiled code with a fixed tile size. The arithmetic mean of the overhead across all matrix sizes is 0.9%.

**Figure 11** The tiled five-point SOR stencil code used in the experiments

```

for (jj = 1; jj < nn; jj += tj)
  for (ii = 1; ii < nn; ii += ti)
    for (i = ii; i < MIN(nn-1, ii+ti); i++)
      for (j = jj; j < MIN(nn-1, jj+tj); j++)
        a[i*nn + j] = 0.2*(a[i*nn+j]+a[(i+1)*nn+j]+a[(i-1)*nn+j]+
          a[i*nn+j+1] + a[i*nn+j-1]);

```

**Figure 12** The tiled LU code used in the experiments

```

for (ii = 0; ii < nn; ii += ti)
  for (jj = 0; jj < nn; jj += tj)
    for (k = 0; k < nn; k++)
      for (i = MAX(k+1, ii); i < MIN(nn, ii+ti-1); i++)
        for (j = MAX(k+1, jj); j < MIN(nn, jj+tj-1); j++) {
          if ((jj<=k + 1) && (k + 1 <= jj+ tj -1) && (j == MAX(k+1, jj)))
            a[i*nn+k] /= a[k*nn+k];
          a[i*nn+j] -= a[i*nn+k]*a[k*nn+j]; }

```

The positive effect of dynamic tiling can be observed by comparing the fifth, sixth and seventh bar in each set (four threads running on two processors) and the 11, 12 and 13 bar in each set (eight threads running on four processors). The execution time savings by using dynamic tiling and copy for four threads running on two processors average 26%.<sup>3</sup> For eight threads running on four processors dynamic tiling with copy improves performance by 23%. Dynamic tiling using the block layout improves performance slightly less than dynamic tiling with copy, 24% with four threads running on two processors and 20% with eight threads running on four processors.

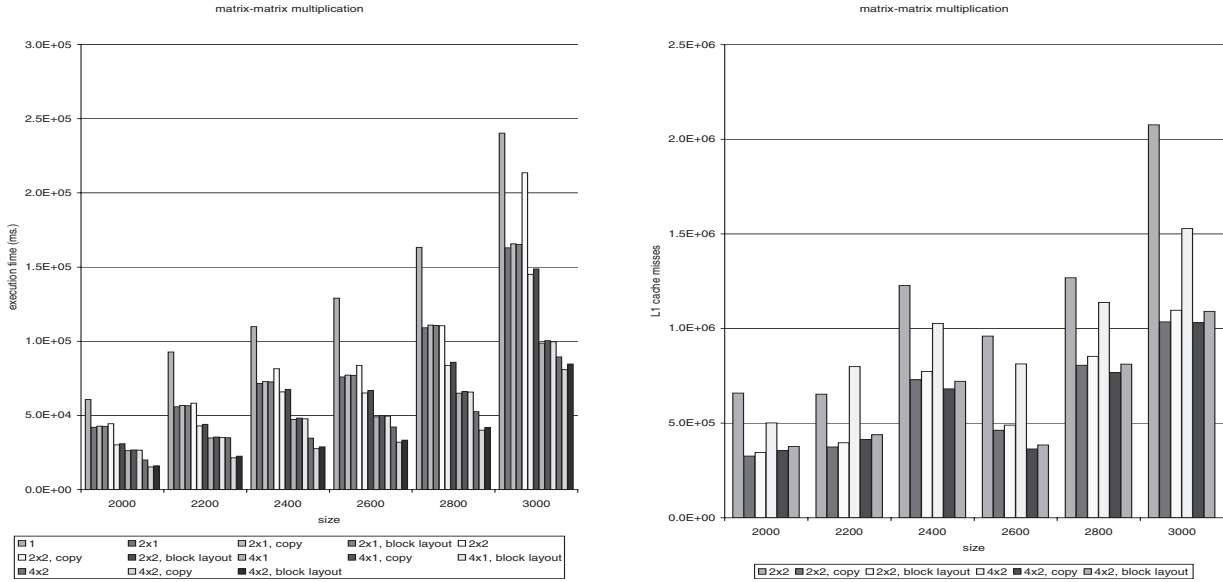
With dynamic tiling and copy, the average speedup is improved from 1.41 to 1.91 when two processors run four threads and from 2.95 to 3.89 when four processors run eight threads. The sublinear speedups are attributed to load imbalance due to uneven distribution of tiles between threads, memory hierarchy effects, including limitations of bus bandwidth and bus contention, and contention for shared resources between threads running on the same processor. The load imbalance problem is addressable, but its solution is beyond the scope of this paper. With the block layout, the average speedup is at 1.86 when two processors run four threads and 3.72 when four processors run eight threads. It is interesting to note that using the block layout, we can get more than 95% of the performance we get by using copy. This result is encouraging in the sense that using block layouts is a more generally applicable transformation, as discussed in Section 2.

A closer examination of the results for  $m \times m$  reveals that simultaneous multithreading on a single processor does not always improve performance. With fixed tile sizes, using four threads on two processors leads to a performance drop of 9%, compared with the performance achieved with two threads running on two processors without sharing the data cache. On the contrary, with dynamic tiling, performance is improved by 18% in the same setting. With eight threads on four processors, performance is always improved, compared with the performance achieved with four threads running on four processors. The improvement is 15% with fixed tile sizes. It jumps to more than 35% with dynamic tiling.

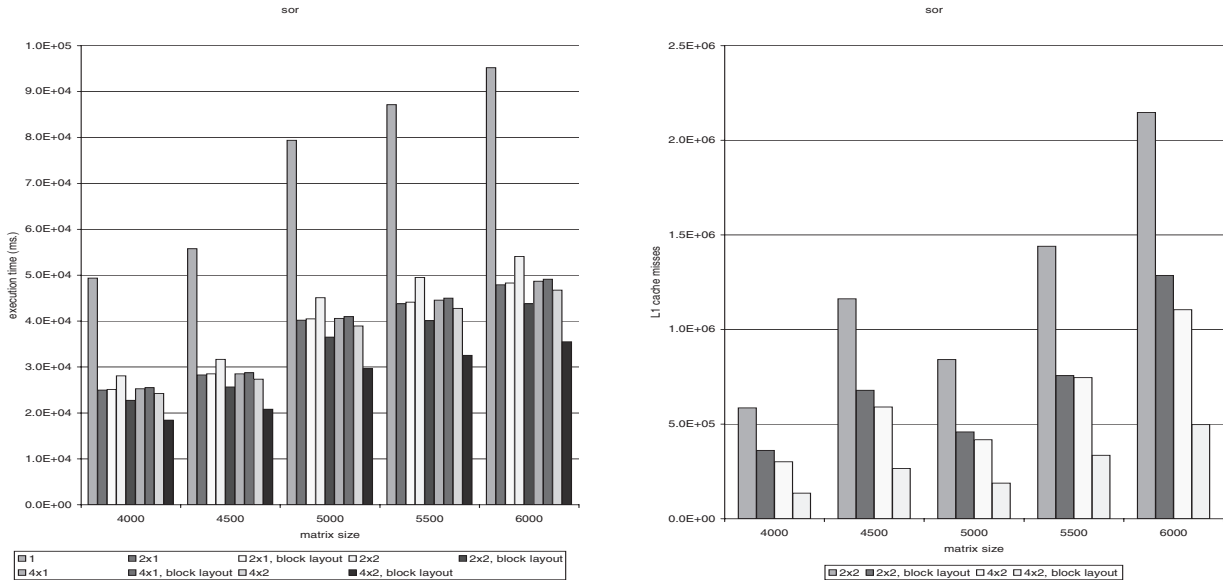
The right chart in Figure 13 shows the number of L1 cache misses throughout the execution of  $m \times m$  using fixed tile sizes, dynamic tiling combined with copy and dynamic tiling combined with block array layouts. We show only results from experiments in which the cache is shared (four threads on two processors and eight threads on four processors). Each bar is the arithmetic mean of the number of cache misses across the threads, during the execution of the tiled loop nest. On two processors running two threads each, L1 cache misses are reduced by 53%. On four processors running two threads each, L1 cache misses are reduced by 61%. L2 cache misses (not shown in the charts) are reduced by 56% and 60%, respectively. Although dynamic tiling reduces cache misses by a wider margin when eight threads are executing simultaneously, the parallel execution time is not improved accordingly. A more careful analysis of the results indicates that there is a significant amount of execution time wasted in the OpenMP barrier at the end of the loop when eight threads were used. This pin points load imbalance and not data locality as the problem.

Figure 14 shows the execution times and L1 cache misses of the tiled SOR code. The performance trends are similar to those observed in  $m \times m$ . Using block layouts introduces negligible overhead if threads do not share the cache. The overhead ranges from 0.7% to 0.9%. Recall that the SOR can not be perfectly tiled using the block layout, because we need to peel off the iterations that access the elements around the boundary of a block and access these elements with modulo and division operations. Despite this overhead, execution time is improved by 19% when four threads run on two processors and 32% when eight threads run on four processors. Speedup is improved from 1.76 to 2.17 with four threads on two processors and from 2.07 to 2.68 with eight threads on four processors. Using two threads instead of one thread on the same processor leads to marginal performance losses or improvements with fixed tile sizes (in the range of  $\pm 5\%$ ) but significant improvements (ranging from 12% to 32%) with dynamic tiling. L1 cache misses are reduced by 54%. L2 cache misses (not shown in the charts) are reduced by 53%.

**Figure 13** Execution times (left chart) and number of L1 cache misses (right chart) of an  $n \times n$ ,  $2000 \leq n \leq 3000$ ,  $m \times m$  with fixed tile sizes, dynamic tiling with copy and dynamic tiling combined with block array layouts



**Figure 14** Execution times (left chart) and number of L1 cache misses (right chart) of the tiled SOR code using arrays of size  $n \times n$ ,  $4000 \leq n \leq 6000$ . Results are reported for one version using fixed tile sizes and another using dynamic tiling and the block layout for  $a$



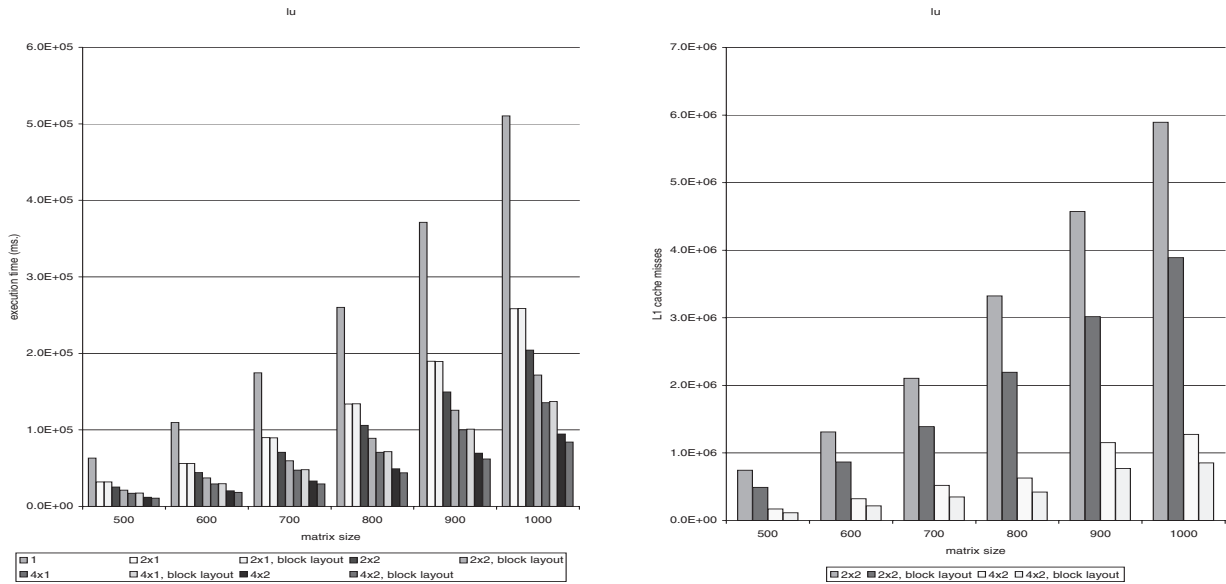
The results for LU (Figure 15) do not differ qualitatively with the results obtained with the other two codes. However, the performance gains obtained from dynamic tiling and block layouts are noticeably reduced to 10–15%. This is explained by the fact that most array accesses in LU cannot be optimised to discard the modulo and division operations for indexing the array. Other than the write access to  $a$  on the LHS of the second assignment in the loop body (Figure 12), all other accesses are not translated and use the non-optimised indices.

Figure 16 shows the average execution time of two tiled  $m \times m$  running simultaneously with four threads each, using a stride-2 allocation of hardware contexts to threads. This means that the first  $m \times m$  runs on even-numbered hardware contexts and the second  $m \times m$  runs on odd-numbered

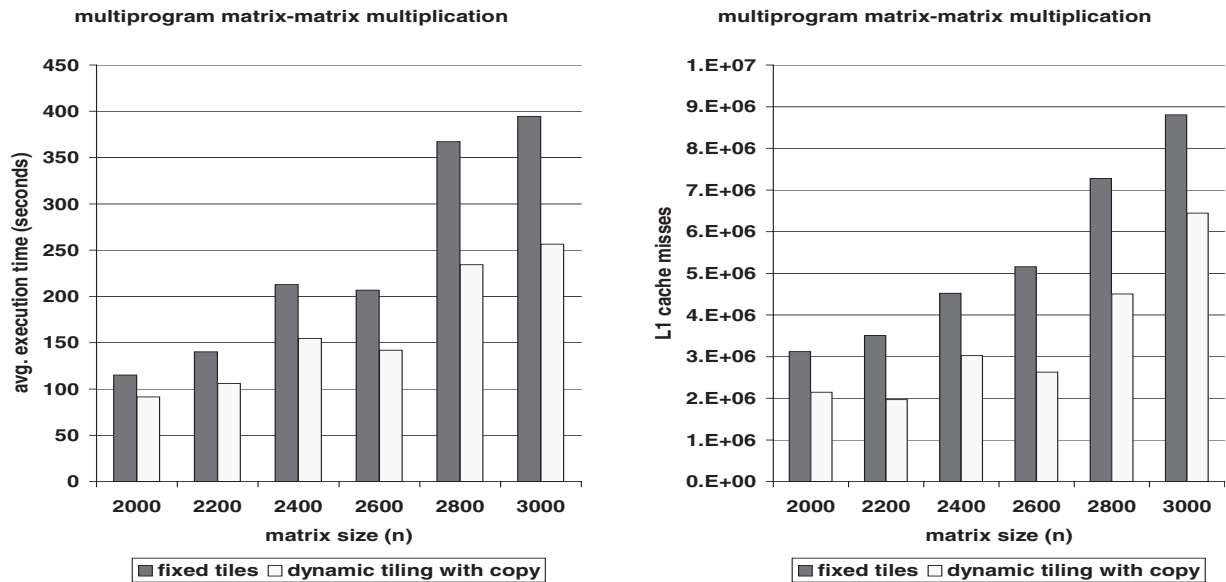
hardware contexts. This experiment evaluates if dynamic tiling can alleviate cache contention between independent tiled codes sharing the cache in a multiprogram workload. As expected, the results are very similar to the ones obtained when a single tiled program is using threads that share the data cache, as both programs activate and use dynamic tiling with copy independently. Execution time is reduced by 29%. L1 cache misses are reduced by 65% and L2 cache misses (not shown) are reduced by 74%.

We ran one last experiment to test if our techniques work equally well with a more dynamic multiprogram workload competing for cache space. We executed the tiled  $m \times m$  with four threads, using a stride-2 allocation of threads to hardware contexts. Concurrently, we launched a script with a blend of

**Figure 15** Execution times (left chart) and number of L1 cache misses (right chart) of the tiled LU code, using arrays of size  $n \times n$ ,  $600 \leq n \leq 1000$ . Results are shown for two versions of the code, one using fixed tile sizes and one using dynamic tiling and the block layout for  $a$



**Figure 16** Average execution time and number of L1 cache misses of two tiled  $m \times m$  executed with four threads each, using stride-2, odd-even allocation of hardware contexts to threads

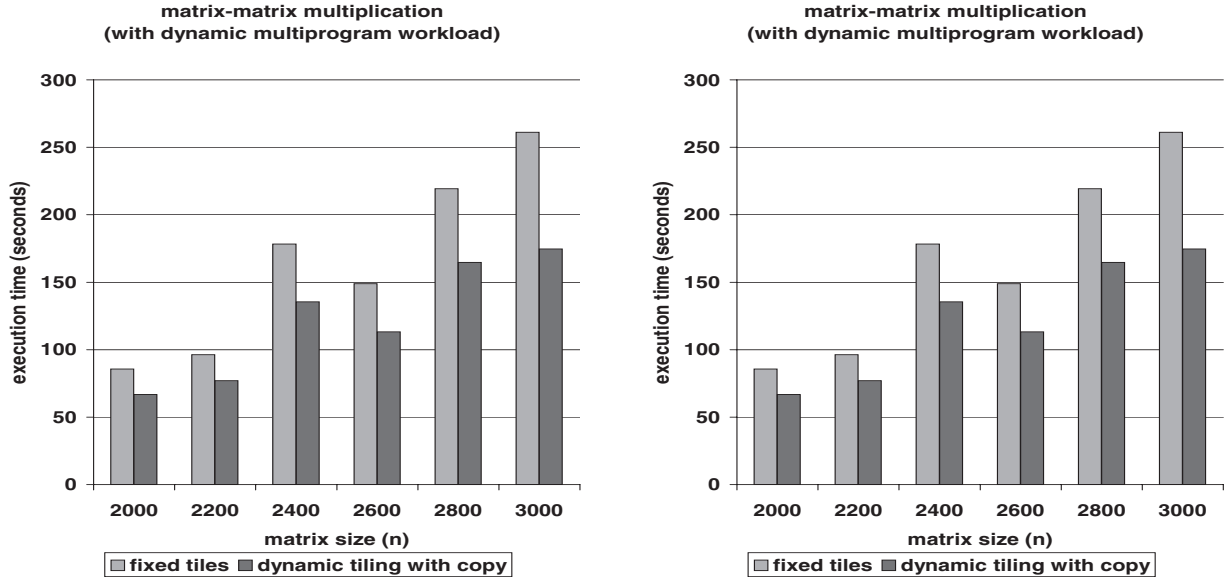


integer, floating point, system call, disk I/O and networking microbenchmarks. This script is adapted from Caldera’s AIM benchmark for Linux systems (Version s9110). The script was modified so that individual microbenchmarks were distributed between the odd-numbered virtual processors, while the even-numbered virtual processors were executing the parallel  $m \times m$ . This experiment creates significant load imbalance between the processors, as some of the benchmarks (e.g. I/O and networking) make intensive use of the cache due to buffering, while others (e.g. system calls) make little or no use of the cache.

Figure 17 shows the execution time and L1 cache misses of  $m \times m$  in this workload. Performance is improved by 28%.

L1 cache misses are reduced by 70%. During the experiments, we instrumented the code to measure the fraction of time during which the code was executing with the reduced tile size. We used cache miss rates per reuse iteration as the indication of cache contention. A smaller tile size was used 85% of the time. The code switched four times from bigger to smaller tiles and vice versa. Using performance counters to detect contention made the code more adaptive than using `top` or notifications from the OS through the `/proc` filesystem. In the latter case, the code switched only twice and a smaller tile size was used 95% of the time. This resulted to slightly lower performance for the tiled code.

**Figure 17** Execution time and L1 cache misses of one tiled  $m \times m$  with four threads running on even-numbered hardware contexts, while a dynamic multiprogram workload is running concurrently on the four odd-numbered hardware contexts



To summarise, the experiments have shown that the transformations proposed in this paper handle well contention in the data cache due to parallel or multiprogram execution on a SMT multiprocessor. Dynamic tiling with copy and dynamic tiling combined with block array layouts are effective optimisations for SMT processors running in multithreaded mode. Using block layouts is a more generally applicable method; however, its performance may be somewhat constrained by the overhead of creating the layouts and the fact that not all array accesses in a loop nest can be optimised by eliminating the expensive indexing operations. Dynamic tiling with copy is effective if the tiled code uses primarily read-only blocks and each block exhibits significant reuse. It is possible to combine these transformations in the same code, as well as apply the transformations selectively if in certain cases their overhead outweighs the benefit from reducing cache misses.

## 5 Related work

SMT processors have been a focal topic in computer architecture research for almost 10 years, initially in the context of microprocessor design (Lo et al., 1997; Tullsen et al., 1995) and later, in the context of system software design (McDowell et al., 2003; Redstone et al., 2000; Snively and Tullsen, 2000).

This paper is among the first to propose and measure the impact of transparent program optimisations targeting specifically SMT processors. It is also among the first to use an actual commercial SMT processor for this purpose. A recent study conducted concurrently with this work (Tuck and Tullsen, 2003) has investigated the processor that we used (Intel Pentium4 with Hyperthreading support) in a broader context, using several microbenchmarks and standard benchmark suites, such as SPEC and Splash. This study is more general and investigates the effect of simultaneous multithreading in terms of parallel program

speedup and job throughput. Our study focuses exclusively on one aspect of the design of SMT processors, the shared data cache and the program optimisations required to tame negative effects that stem from this design choice.

The work of Lo and others (Lo et al., 1997) is the most closely related to the work presented in this paper. The authors investigated whether standard compiler techniques for loop iteration scheduling, software speculative execution and loop tiling should be reconsidered and adapted to SMT processors. This work did not consider multiprogram workloads or adaptation to both single-threaded and multithreaded execution modes. For loop tiling in particular, this study has observed that the effect of using simple tiling and assign whole tiles to individual threads can be detrimental for performance. The study recommended using cyclic tiling on each multithreaded processor to solve the problem of conflicts between tiles. The idea of cyclic tiling is to share the iterations of a tile between threads that run on the same processor.

Cyclic tiling yields good performance for a single multithreaded processor but on a multiprocessor of multithreaded processors, a cyclic allocation of the iterations that access a tile forces communication. Table 1 reports the execution time of a tiled  $3000 \times 3000$   $m \times m$  using cyclic tiling and dynamic tiling with copy. For cyclic tiling to be effective, it needs to be restricted within SMT processors and combined with a block or block-cyclic assignment of tiles to threads across the multithreaded processors. Furthermore, the tiles themselves must be hierarchically decomposed. As shown in Table 1, if this hierarchical tiling scheme is used, the performance is improved to within 14% of the performance of the dynamic tiling transformations. Adaptive tiling still gains in performance because compared to hierarchical tiling it lowers the overhead of the tile control code and allows the flexibility to use the optimal tile size for either single-threaded or multithreaded mode, without always forcing the program to nest tiles.

**Table 1** Comparison between dynamic tiling with copy, cyclic tiling and hierarchical tiling. The table reports the execution time of a  $3000 \times 3000$   $m \times m$  using the different tiling schemes

Cyclic tiling	109.46
Dynamic tiling with copy	80.96
Hierarchical tiling	93.56

Tiling and multilevel blocking of array codes for sequential and parallel machines have been investigated in a large number of publications, only a sample of which is referenced here (Coleman and McKinley 1995; Chame and Moon, 1999; Kodukula et al., 1997; Mateev et al., 2000; McKinley et al., 1996; Rivera and Tseng, 1999; Temam et al., 1993; Wolf and Lam, 1991, Xue, 2000). None of these studies considered multithreaded processors. Interestingly enough, (Lo et al., 1997) reported that optimising the tile size does neither hurt, nor improve the memory hierarchy performance of SMT processors, due to the inherent latency hiding capabilities of the processor. This could be true for the small L1 caches. However, for the relatively large L2 and L3 caches choosing a proper tile size is important. We still observe sizeable difference in performance with different tile sizes in our experiments, even if tiling is restricted to the L1 cache.

Block copy and static analysis methods to select between copy and simple tiling were discussed extensively (Temam et al., 1993). The performance advantages of using block layouts in modern memory hierarchies were discussed (Park et al., 2002). A thorough formal treatment for deriving block data layouts of arrays in scientific codes was presented (Gustavson, 2003).

Recently, several researchers have turned their attention to managing shared caches on multithreaded processors and chip multiprocessors running multiprogram workloads, via hardware and software techniques, such as static cache partitioning (Suh et al., 2001) and memory-aware job scheduling (Suh et al., 2002). These studies place the burden of effective management of shared caches on the operating system, or recommend hardware cache partitioning schemes to reduce cache conflicts.

## 6 Conclusion

We presented standard program transformations and runtime dynamic tiling to reduce data cache conflicts between threads on multithreaded processors. This work targets scientific codes which are tiled to maximise cache utilisation but their parallel performance on multithreaded processors is harmed by conflicts between threads in the shared data cache. We have shown that the proposed transformations resulted to sizeable performance improvements over naively tiled codes. More importantly, we have shown that with the proposed transformations, SMT processors can be used effectively for running parallel codes with substantial speedup. The existing research favours multithreaded processors for running multiprogram workloads but concludes that these processors are somewhat less effective in the execution of parallel programs. This work is a step towards improving

the performance of parallel programs on multithreaded processors, without necessarily sacrificing the capability of the processor to improve throughput in multiprogram workloads.

There are several directions for future work in the context of improving parallel program execution on multithreaded processors. This study did not conduct a thorough investigation of the effects of different tile scheduling algorithms on SMT multiprocessors. This is a problem which requires immediate attention, as poor scheduling may nullify the gains of effective tiling. We have primarily studied the performance of the data cache but for certain workloads and large codes it is necessary to study the performance of the TLB as well, as the TLB is another shared hardware resource on SMT processors. Besides the runtime mechanisms presented in this paper, we would like to develop a convenient analytical framework for quantifying conflict misses in shared caches and use this framework to derive automatic program transformations statically. 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.

## Acknowledgement

The authors would like to thank the IJHPCN referees for several helpful suggestions. A preliminary version of this work has appeared (Nikolopoulos, 2003).

## References

- Anderson, J., Amarasinghe, S. and Lam, M. (1995) 'Data and computation transformations for multiprocessors', in *Proceedings of the Fifth ACM Symposium on Principles and Practices of Parallel Programming (PPoPP'95)*, Santa Barbara, California, July, pp.166–178.
- Browne, S., Dongarra, J., Garner, N., London, K. and Mucci, P.A. (2000) 'Scalable cross-platform infrastructure for application performance tuning using hardware counters', in *Proceedings of Supercomputing'2000: High Performance Networking and Computing Conference*, Dallas, TX, November.
- Carter, L., Ferrante, J. and Hummel, S. (1995) 'Hierarchical tiling for improved superscalar performance', in *Proceedings of the Ninth International Parallel Processing Symposium (IPPS'95)*, Santa Barbara, CA, April, pp.239–245.
- Coleman, S. and McKinley, K. (1995) 'Tile size selection using cache organisation and data layout', in *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'95)*, San Diego, CA, June, pp.279–290.
- Carr, S. and Lehoucq, R. (1997) 'Compiler blockability of dense matrix factorisations', *ACM Transactions on Mathematical Software*, Vol. 23, No. 3, pp. 336–361.
- Chame, J. and Moon, S. (1999) 'A tile selection algorithm for data locality and cache interference', in *Proceedings of the 13th ACM International Conference on Supercomputing (ICS'99)*, Rhodes, Greece, June, pp.492–499.
- Cross, R. (2002) 'Intel hyper-threading technology', *Intel Technology Journal*, Vol. 6, No. 1.

- Gustavson, F. (2002) 'High performance linear algebra algorithms using new generalized data structures and matrices', *IBM Journal of Research and Development*, Vol. 17, No. 1, pp.31–56.
- Kodukula, I., Ahmed, N. and Pingali, K. (1997) 'Data-centric multilevel blocking', in *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'97)*, Las Vegas, Nevada, June, pp.346–357.
- Lo, J., Emer, J., Levy, H., Stamm, R., Tullsen, D. and Eggers, S. (1997) 'Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading', *ACM Transactions on Computer Systems*, Vol. 15, No. 3, pp.322–353.
- Lo, J., Eggers, S., Levy, H., Parekh, S. and Tullsen, D. (1997) 'Tuning compiler optimisations for simultaneous multithreading', in *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO'30)*, Research Triangle Park, NC, November, pp.114–124.
- McKinley, K., Carr, S. and Tseng, C. (1996) 'Improving data locality with loop transformations', *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 4, pp.424–453.
- Mateev, N., Ahmed, N. and Pingali, K. (2000) 'Tiling imperfect loop nests', in *Proceedings of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC'2000)*, Dallas, TX, November.
- McDowell, L., Eggers, S. and Gribble, S. (2003) 'Improving server software support for simultaneous multithreaded processors', in *Proceedings of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'2003)*, San Diego, CA, June.
- Nikolopoulos, D.S. (2003) 'Code and data transformations for improving shared cache performance on SMT processors', in *Proceedings of the Fifth International Symposium on High Performance Computing (ISHPC-V)*, Tokyo, Japan, October.
- Park, N., Hong, B. and Prasanna, V. (2002) 'Analysis of memory hierarchy performance of block data layout', in *Proceedings of the 2002 International Conference on Parallel Processing (ICPP'2002)* Vancouver, Canada, August, pp.35–42.
- Rivera, G. and Tseng, C. (1999) 'A Comparison of tiling algorithms', in *Proceedings of the Eighth International Conference on Compiler Construction (CC'99)* Amsterdam, the Netherlands, March, pp.168–182.
- Redstone, J., Eggers, S. and Levy, H. (2000) 'Analysis of operating system behavior on a simultaneous multithreaded architecture', in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IX)*, Cambridge, MA, November.
- Snively, A. and Tullsen, D. (2000) 'Symbiotic job scheduling for a simultaneous multithreading processor', in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'IX)*, Cambridge, Massachusetts, November, pp.234–244.
- Suh, G., Devadas, S. and Rudolph, L. (2001) 'Analytical cache models with applications to cache partitioning', in *Proceedings of the 15th ACM International Conference on Supercomputing (ICS'01)*, Sorrento, Italy, June, pp.1–12.
- Suh, G., Rudolph, L. and Devadas, S. (2002) 'Effects of memory performance on parallel job scheduling', in *Proceedings of the eighth Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'02)*, Edinburgh, Scotland, June, pp.116–132.
- Temam, O., Granston, E. and Jalby, W. (1993) 'To copy or not to copy: a compile-time technique for assessing when data copying should be used to eliminate cache conflicts', in *Proceedings of the ACM/IEEE Supercomputing'93: High Performance Networking and Computing Conference (SC'93)*, Portland, OR, November, pp.410–419.
- Tullsen, D., Eggers, S. and Levy, H. (1995) 'Simultaneous multithreading: maximizing on-chip parallelism', in *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA'95)*, St. Margherita Ligure, Italy, June, pp.392–403.
- Tuck, N. and Tullsen, D. (2003) 'Initial observations of the simultaneous multithreading pentium IV processor', in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'2003)*, New Orleans, Louisiana, September, pp.26–35.
- Wolf, M. and Lam, M. (1991) 'A Data locality optimizing algorithm', in *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, Toronto, Canada, June, pp.30–44.
- Xue, J. (2000) *Loop Tiling for Parallelism*, Kluwer Academic Publishers.

## Notes

- <sup>1</sup> Currently, Linux assigns different processor IDs to the hardware threads on an SMT processor, as if each hardware thread were a CPU on its own.
- <sup>2</sup> In Linux, each kernel thread has a unique identifier.
- <sup>3</sup> All reported averages are arithmetic means taken from the executions of a program with a fixed number of threads across all data set sizes.