Journal of Universal Computer Science, vol. 6, no. 10 (2000), 968-993
submitted: 1/4/00, accepted: 13/9/00, appeared: 28/10/00 © Springer Pub. Co.

Compiler Generated Multithreading to Alleviate Memory Latency

Kristof E. Beyls
(Dept. of Electronics and Information Systems
University of Ghent, Belgium
kbeyls@elis.rug.ac.be)

Erik H. D'Hollander
(Dept. of Electronics and Information Systems
University of Ghent, Belgium
dhollander@elis.rug.ac.be)

Abstract: Since the era of vector and pipelined computing, the computational speed is limited by the memory access time. Faster caches and more cache levels are used to bridge the growing gap between the memory and processor speeds. With the advent of multithreaded processors, it becomes feasible to concurrently fetch data and compute in two cooperating threads. A technique is presented to generate these threads at compile time, taking into account the characteristics of both the program and the underlying architecture. The results have been evaluated for an explicitly parallel processor. With a number of common programs the data-fetch thread allows to continue the computation without cache miss stalls.

Key Words: data locality, multithreading, run-time data relocation, compiler optimization, cache optimization, prefetching, tiling

1 Introduction

The well known Von Neumann bottleneck has hampered the unbridled development of shared memory multiprocessors. New parallel programming paradigms, architectures and interconnection networks with huge bandwidth have increased the computing power, but the memory latency remains the limiting factor of the computation. Whereas the processor performance increases by about 60%/year, the memory performance increases only by about 7%, leading to a relative performance drop of 2 every 21 months. In the beginning, a single cache level sufficed to dampen the effect of slow memory access, but the widening gap has led to a hierarchy of caches. This cache hierarchy limits the damage, but clearly the memory latency is important when cache misses occur. It is not uncommon that a processor stalls for memory accesses more often than doing useful computations. With more data-hungry superscalar and multithreaded processors, this trend is not likely to change.

To tighten the processor-memory gap, several hardware and software techniques have been proposed to use the cache hierarchy more effectively and to hide the memory latency:

1. Software techniques which reorder the program loop structure in order to promote data locality and reuse, such as loop permutation, reversal, fusion, distribution[McKinley *et al.*1996] and tiling[Lam *et al.*1991].
2. Software and hardware techniques which improve the usage of the limited cache capacity, such as skewed-associative caches[Seznec *et al.*1993], victim caches[Jouppi1990], array padding[Rivera *et al.*1998], data copying at run-time[Temam *et al.*1993], ...
3. Techniques to hide the latency of inevitable cache misses with useful computations such as prefetching[Baer *et al.*, Mowry *et al.*1992], multithreading[Weber *et al.*1989, Alverson *et al.*1995, Mowry *et al.*1998], non-blocking caches, dynamic instruction scheduling, ...

Modern processor architectures expose the cache to the software by introducing prefetch instructions and cache hints in their instruction sets. It allows the programmer to assist the cache hardware in making decisions about what and when to cache. In this paper, these cache instructions are used to implement *cache remapping*, a new technique which optimizes cache behavior. Cache remapping uses the multithreading capabilities of a processor to pipeline the memory access and to improve cache usage. The first step is to create a separate data-fetch thread,which continuously fetches the data needed in future computations. The second step is to store the data in a free area of the cache The third step is to use the fetched data in the new calculations. A well balanced, multithreaded program will experience only a few cache misses at start-up. For scientific loop kernels, a compiler can automatically perform the necessary transformations to implement this technique.

In the following section, a formal cache model is introduced to explain and analyze the caching policy used in cache remapping. In [Section 3] the data placement, thread-synchronization and architecture requirements necessary to implement cache remapping are discussed. The compiler implementation is discussed in [Section 4]. In [Section 5] the performance of cache remapping is compared with optimized non-threaded execution. Speedups using multithreading yield significant improvement over the best existing single-threaded version.

2 Modeling Cache and Locality

2.1 Cache Model

The cache is characterized by a 4-element tuple (C_s, N_s, k, L_s), cache size, number of sets, associativity and line size respectively[Ghosh1999]:

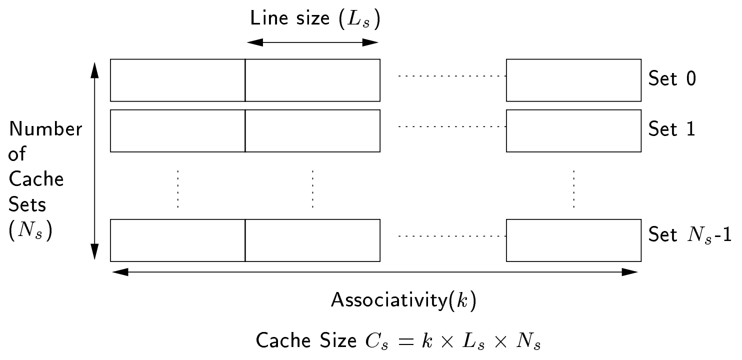

Line size (L_s)
Set 0
Set 1
Set N_s-1
Number of Cache Sets (N_s)
Associativity(k)
Cache Size $C_s = k \times L_s \times N_s$

Figure 1: Illustration of cache organization

The *cache size (C_s)* is the total number of data elements in the cache. The *line size (L_s)* is the number of adjacent data elements fetched on a cache miss. A *memory line* refers to a cache-line-sized block in the memory, aligned to a single cache line. A *cache set* is a collection of cache lines which can hold a particular memory line. N_s denotes the number of cache sets in a cache. *Associativity (k)* refers to the number of cache lines in a cache set. These parameters are related by the equation $C_s = N_s \times k \times L_s$.

Cache misses occur when an addressed memory line is not in the cache. Cache misses can be categorized by the way cache lines are overwritten[Hill *et al.*1989]:

cold misses occur when data is referenced for the first time.

capacity misses occur when to many other references since the previous reference to the same data have occured. The cache is just to small to enable the reuse.

conflict misses result from the fact that in caches with a limited associativity, data at a given address can only be stored in a limited number of cache lines. If the miss wouldn't have occured in a fully associative cache of the same size with the LRU replacement policy, it is a conflict miss. By definition, conflict misses do not occur in fully-associative caches with LRU replacement.

The cache set N, holding the memory line starting at address A, is given by:

$$N = \left\lfloor \frac{A}{L_s} \right\rfloor \bmod N_s \tag{1}$$

A replacement algorithm decides which line in set N is overwritten by memory line $\left\lfloor \frac{A}{L_s} \right\rfloor$.

2.2 Caching Policy

A *caching policy* is a mechanism for processing the memory reference string $\omega = r_1, r_2, \ldots, r_t, \ldots$ resulting from a program execution[Coffman *et al.*1973]. Let $Adr(r_i \ldots r_j)$ be the set of *different* data addresses accessed by references $r_i \ldots r_j$. The caching policy implemented by cache remapping is as follows:

- First, the reference string ω is partitioned into subsequences $P_1^\omega P_2^\omega \ldots$ such that each partition can be stored in one-half of the available data cache size C_s:
$$\frac{C_s}{2} \geq |Adr(P_i^\omega)| \tag{2}$$
- With every trace partition P_i^ω a code fragment C_i^ω is associated. Initially the data for partition P_0^ω is fetched into the cache. Next in phase i concurrently the data for partition P_{i+1}^ω is prefetched and the code C_i^ω is executed. Due to cache remapping, prefetching and execution operate on different regions in the cache.

The partitioning of the reference string implicitly implies a partitioning of the program into code blocks. This requires a program analysis and transformation as well as a multithreaded execution kernel. The program transformation and compiler implementation will be further discussed in [Section 4]. The implementation of software-directed caching policies in general and the above caching policy in particular are further discussed in detail in [Section 3].

2.3 Prefetch Cost

In order to achieve a smooth and continuous execution after the initial prefetch of the first partition, the computation time, CT, should always exceed the prefetch time, FT, so
$$CT(P_i^\omega) \geq FT(Adr(P_{i+1}^\omega)).$$

To satisfy this condition, it is important to reuse the data of a code block as much as possible by a suitable restructuring of the computations. In the next subsection, the importance of reuse is derived theoretically from the cost function of the caching algorithm. After that, an example is given where the restructuring of the program leads to improved performance by increasing the reuse.

2.3.1 Cost Function and Reuse Factor

The goal of a caching algorithm is to minimize the time needed to process a certain reference string ω. The cost function indicates how many cache misses the program with reference string ω experiences, given a certain caching policy.

The cache misses/memory-cache data transfers that do not hinder the execution of the program, e.g. when prefetching, are not added up to the cost function. $FT(Adr(P_i^\omega))$ denotes the time needed to fetch the unique data elements referenced in P_i^ω into the cache. $CT(P_i^\omega)$ gives the time needed to execute C_i^ω, the code block associated with P_i^ω. The cost function for the cache remapping caching policy is:

$$\begin{aligned} C(\omega, C_s) = {} & FT\left(Adr(P_0^\omega)\right) \\ & + \sum_{P_i^\omega} \max\left(0, FT\left(Adr(P_{i+1}^\omega)\right) - CT(P_i^\omega)\right) \end{aligned}$$

The first term indicates the time needed to fetch the data used in the first trace partition, which is not overlapped by useful computation. The last term is the summation of all trace partition fetching not overlapped with useful computations. The cost can be reduced by minimizing $FT(Adr(P_{i+1}^\omega)) - CT(P_i^\omega)$. Reusing the same data in as many useful computations as possible leads to an increased P_i^ω with a constant $Adr(P_{i+1}^\omega)$. So, increasing the reuse factor, defined as the average number of accesses to the same element in a partition, decreases the cost.

2.3.2 Reuse-Enhancing Program Transformation

The above caching policy could in principle be applied to any program. However, to make the code partitioning implementable in a compiler, we focus on loops kernels, which typically account for most execution time in scientific code.

Most program transformation theory is focused on array-based normalized perfect loop nests with affine array index expressions and affine loop boundaries. In this section, we build on this theory to maximize the reuse factor inside a trace partition. First, a couple of definitions about array-based perfect loop nests are given.

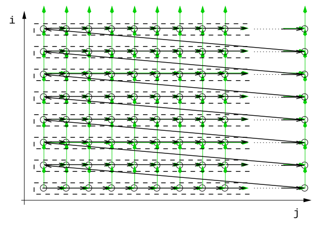
The *iteration space*[Irigoin *et al.*1988] of an n-deep loop nest is a polytope in $\mathcal{Z}^n$ bounded by the loop boundaries. Every point in the iteration space represents the execution of a single iteration. The iteration space can be plotted in an iteration space graph, as can be seen in [Fig. 2]. The loop nest dictates in which order the program traverses the iteration space.

Reuse vectors[Mowry1994] indicate the reuse of data between iterations. Only temporal reuse vectors are considered here. A temporal reuse vector is the iteration space distance between the use and the reuse of the same data element. In [Fig. 2] for example, the temporal reuse vectors are $(0, 1)$ and $(1, 0)$.

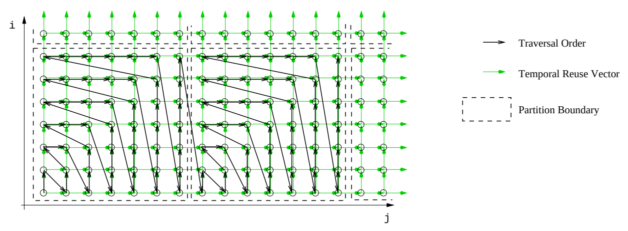
Reuse vectors define a distance in the iterations space, but here the distance of reuse in the reference string ω is more interesting. If to much data is accessed (and cached) in between the use and the reuse of a word, the cache cannot

```
DO i=1,N
   DO j=1,N
      A(i,j) = A(i-1,j) + A(i,j-1)
```

(a) The 2-deep loop nest



(b) The iteration space traversal defined by the loop nest.



(c) The reordered iteration space traversal, resulting in 58% more computations per partition.

Figure 2: An example of a 2-deep loop nest, together with the original iteration space traversal and the iteration space traversal with improved reuse characteristics. The iterations that are executed in the same partition are surrounded by a dotted box. The reordered iteration space traversal results in shortening the fetch time with 58% relative to the computation thread.

retain that word until it is reused. The reuse vectors don't immediately give the number of intervening references between the use and the reuse of a memory word. Therefore, the *reference distance*[Pyo *et al.*1997] of an element of ω is introduced, which is the number of references since the last use of the same data element.

The goal of reordering the iteration execution is to produce a modified reference string ω with shorter reference distances, enabling more reuse inside a trace partition. It increases the reuse factor because of the equation $|Adr(P_i^\omega)| + \#\text{reuses in } P_i^\omega = |P_i^\omega|$.

To increase reuse inside a partition, the iterations which are linked together by reuse vectors should be executed in the same partition as much as possible. To

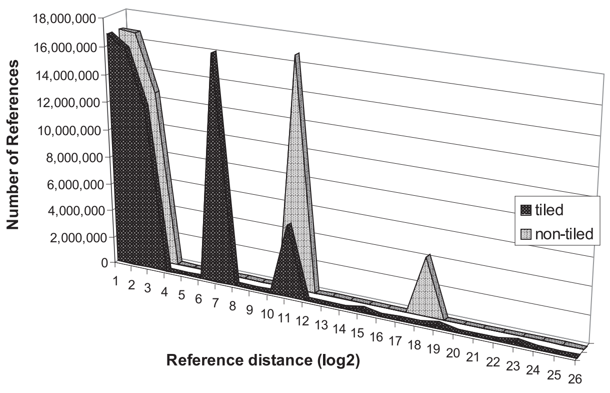


Figure 3: The distribution of reference distances of the matrix multiplication ($N = 256$) before and after tiling (tile size=20). From the graph, it is clear that tiling has shortened the reference distances; references with distance 2^{11} have distance 2^7 and references with distance 2^{18} have distance 2^{11} after tiling. A closer look at the graph of the tiled execution reveals a small number of references at distance $2^{15}, 2^{19}$ and 2^{23}. These result from the reuse of data in between different trace partitions.

achieve this, the following heuristic to traverse the iteration space of the example in [Fig. 2] is used. First, the first point in the iteration space is executed. Next, the iteration point with the most reuse from previously executed iteration points within the current trace partition is executed. This is repeated until the complete iteration space has been traversed.

If this algorithm is applied to the example in [Fig. 2(a)], with $C_s = 128$, the iteration space traversal in [Fig. 2(c)] is obtained. In the original traversal order, 30 reuse vectors are captured inside a partition. In the optimized traversal, 84 reuse vectors are inside a partition. Because of the increased reuse, 49 consecutive iterations access $63 (\leq \frac{C_s}{2})$ different data elements, the same amount of data as 31 iterations in the original order. As a result, the total number of partitions needed to execute the complete loop nest is only $\frac{31}{49} = 63\%$ of the number of partitions in the original traversal. The reuse factor is increased by 58%.

If the traversal order of the iteration points inside a partition permutes, the reuse factor doesn't change because $Adr(P_i^\omega)$ and $|P_i^\omega|$ still have the same value. When the iterations inside a trace partition are traversed in lexicographical order, the same traversal as resulting from loop tiling[Wolf *et al.*1991] is found. In [Section 4], it is shown how loop tiling and cache remapping can be combined to optimize loop kernels automatically in a compiler. As a last example of how tiling can shorten the reference distance, [Fig. 3] shows the reference distances found when executing the matrix multiplication kernel both before and after tiling. It is clear from this figure that tiling shortens the long reference distances and enables a higher reuse factor.

3 Multithreaded Cache Remapping

Instead of bringing the data in the cache by doing a conventional load or prefetch and letting the hardware decide where the data will be stored in the cache, cache remapping uses the advanced possibilities of new semantics of memory instructions. These new instructions indirectly allow the software to control the data placement in the cache.

Normally, the cache hardware decides which cache location the data will be stored into, depending on the address of the data and the past references. The software directed data placement allows the programmer or the compiler to control the cache location used, leading to new possibilities for optimizations.

3.1 Computation and Communication Threads

The work done by the processor while executing the program is partitioned into two threads as follows:

1. the computation thread, which performs all the computational work in the program;
2. the data-fetch thread which ensures that the computation thread does not experience cache misses.

The data-fetch thread remaps data elements to new addresses, corresponding to a free area in the cache. Consequently, the addresses of the memory references in the computations must also be altered. Since this is done at compile time, no run time overhead is incurred. After the data is fetched into the cache, the computation thread performs the calculations without main memory latency. Finally, the data-fetch thread copies the changed data back to the original locations in main memory.

Parallel operation is obtained by executing the computation and data-fetch threads in a pipelined fashion (see [Fig. 4]). The data relocation allows to take advantage of the full potential of the cache by filling the cache lines completely and eliminating cache conflicts by ensuring that data elements are never placed in the same cache location at the same time. The reduction in conflict miss stall time significantly improves the execution speeds, provided the execution time of the data-fetch thread is smaller than the execution time of the computation thread.

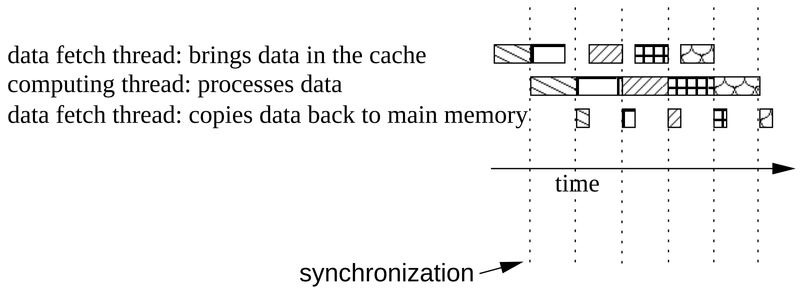

data fetch thread: brings data in the cache
computing thread: processes data
data fetch thread: copies data back to main memory
time
synchronization

Figure 4: The data is pipelined through the data-fetch thread, the computing thread, and the data-fetch thread again. First, the data-fetch thread places it in the cache. Then the computing thread processes the data. Finally, the data-fetch thread copies the changed data back to main memory.

3.2 Data Prefetch and Remapping

3.2.1 Cache Shadow

Even with instruction sets equipped with prefetching instructions and cache hints, the cache cannot be addressed explicitly. However, cache hints can be used to bypass the cache. In order to fully control the data cache, a free main memory region is reserved and maps one-to-one on the cache memory. This free region is called the *cache shadow* (see [Fig. 5]).

The cache remapped code uses the cache shadow as follows:

1. the computation thread operates only on data in the cache shadow;
2. the communication thread moves data between the program data area and the cache shadow;
3. no other memory references address the cache.

Only the communication thread accesses data outside the cache shadow. These data accesses bypass the cache, so they can't remove data already present in the cache. Only data inside the cache shadow is allowed to enter the cache. As a consequence there are no conflicts inside the cache shadow and its contents cannot be evicted from the cache.

The cache shadow can now be used as a very fast local memory. To place data in the cache, it must be copied to an address in the cache shadow. Because it cannot be evicted, the data remains in the cache until it is explicitly overwritten with some other data.

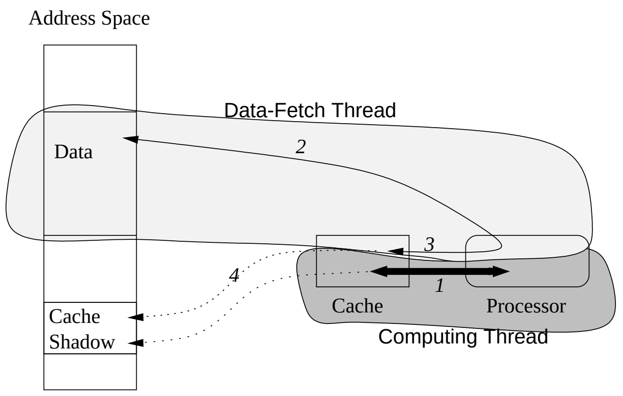


Figure 5: The computing thread only accesses addresses in the cache shadow(*1*). Since the cache shadow is located in the cache, all references are cache hits. The data-fetch thread is responsible for timely copying the data needed by the computing thread into the cache shadow. Cache hints allow the data-fetch thread to access addresses outside the cache shadow without bringing them in the cache(*2*). Once the requested data is in a register, it can be put in the cache by writing it to an address in the cache shadow(*3*). Since the cache is write-back, operations on the cache shadow are not mirrored on the bus(*4*).

3.2.2 Data Placement

Cache hints, emerging in new instruction set architectures[IA64 ADAG1999, Kane1996, Kathail *et al.*2000], are used to control whether load/store instructions cache data or not. The computing thread finds all its data in the cache. The load/store instructions in this thread can have the normal semantics: bring the data in the register and at the same time move the data into the fastest cache level. Because the data is already in the fastest cache level, there will be no data movement in the memory hierarchy besides movements between registers and L1 cache.

The data-fetch thread loads data bypassing the cache using a cache hint, and stores it into the cache shadow, i.e. the fastest cache level (see [Fig. 5]). The load must have different semantics: bring the requested data into a register, but do not move the data into a faster cache level. Cache hints change the semantics of a load instruction to have exactly this meaning. The relocation of a single data element is performed by a code sequence such as the following, using PlayDoh-instructions [Kathail *et al.*2000]:

```
move_into_cache_at_address(int* data_address, int* cache_address)
{
  LD_C3_C3  data_address, Rx  ; load into Rx, bypassing cache
  ST_C1     Rx, cache_address ; store Rx into cache
}
```

`LD_C3_C3` loads the value at address `data_address` into register `Rx`. The first cache hint (`C3`) tells the scheduler that it must assume that the data will be found in cache level 3. The second cache hint (`C3`) states that the loaded data must not be moved closer than cache level 3 (which is the main memory on a system with a 2-level cache). Cache hint `C1` in the store instruction tells the processor that the data should be stored in the L1 cache.

3.2.3 Shadow Memory Partitioning

The computing thread only references addresses in the cache shadow and always hits the cache. The data-fetch thread is responsible for copying variables needed for the computation into the cache shadow.

To avoid conflicts and to facilitate the synchronization between the threads, the cache is partitioned into three parts, P_1, P_2 and P_3 (see [Fig. 6]). P_1 is used to store data that is used and reused throughout the complete program part under consideration, such as constants, induction variables, etc. This data is copied into P_1 at start-up. During the execution of the program, only the computing thread accesses P_1.

The partitions P_2 and P_3 are used alternatively by the computation and communication thread, e.g. while the computation thread uses the data in P_2, the communication thread stores new data into P_3.

As can be seen in [Fig. 4] a kind of pipelined double buffering is applied. The data-fetch thread loads and relocates data for the computing thread into either cache partition P_2 or P_3. After a block of data is copied in the cache, a synchronization follows. After the synchronization, the threads use the other cache partition. Assume the data-fetch thread accesses P_2 and the computing thread accesses P_3. After the next synchronization, the computing thread accesses P_2 and the data-fetch thread accesses P_3. After the synchronization, the computing thread starts processing the data brought into P_2 before the synchronization. The data-fetch thread concurrently frees P_3 by copying the changed values back to their original place in main memory. After the partition is freed, the next block of data is read.

3.3 Synchronization and Alias Resolution

Synchronization between the threads is needed when the computing thread has finished processing all data in its cache partition. Assume that the computing thread was processing the data in P_2. During the processing of the data in P_2

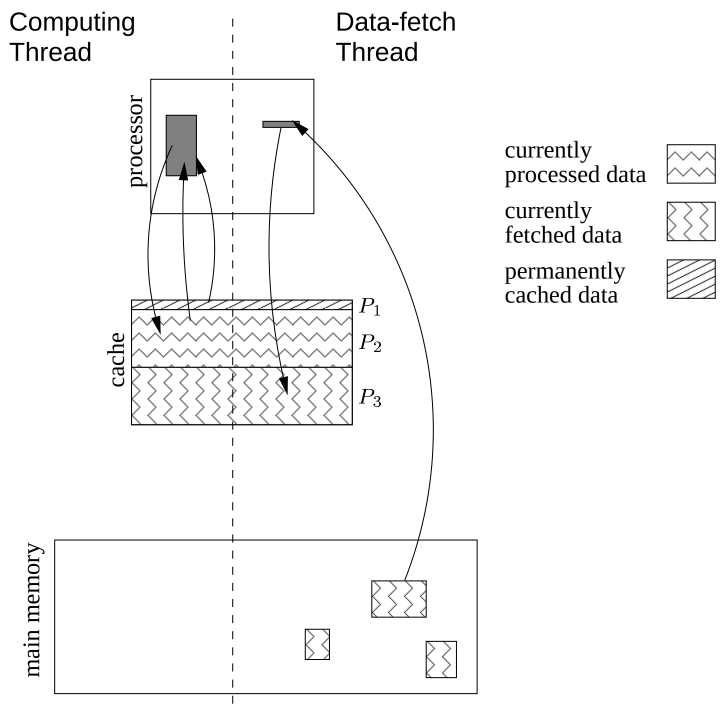


Figure 6: The data-fetch thread puts the next to be used data in one cache partition (P_3) while the computing thread processes the previously cache relocated data, present in P_2. After the next synchronization, the computing and the data-fetch thread will access P_3 and P_2 respectively.

no synchronization is needed because the computing thread only accesses data in P_2 and P_1 (both in the cache) and the data-fetch thread only accesses data in P_3 and the main memory.

Once the computing thread finishes processing the data in P_2, it must start using the next data, placed in P_3 by the data-fetch thread. After the synchronization, the data-fetch thread uses P_2 to place the next data in, while the computing thread uses P_3.

An aliasing problem arises when data dependencies between the computation and the communication thread are not respected. It is possible that the same data object is needed for computations now, and after the next synchronization. As it is needed now, it will be located in the computing cache partition. As it is needed after the next synchronization, the communication thread will fetch

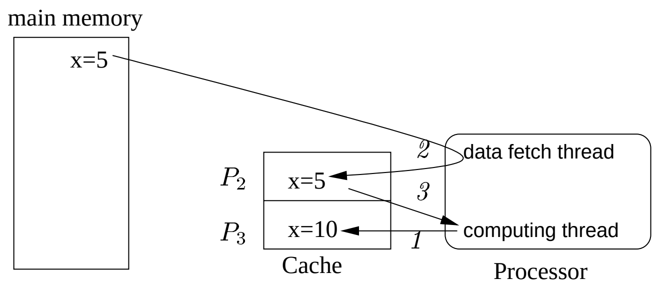


Figure 7: An example of the aliasing problem. The object x is currently being processed by the computing thread(*1*), which assigns it the value 10 (the old value was 5). At the same time (*2*) the data-fetch thread copies the old value (5) of object x in the other cache partition because the computing thread will use that object after the next synchronization. After synchronization, the computing thread will use (*3*) x and read the old and incorrect value 5, instead of the correct value 10.

it from main memory to bring it in the communication partition. However, if the computing thread changes the value of the data object, the communication thread reads the value from main memory before it has been updated and a WAR hazard occurs. An incorrect execution caused by not respecting the data dependency is illustrated in [Fig. 7].

To resolve this problem, the WAR hazards that have occured during the parallel execution of both threads are undone by copying the newest value for the data object from the computation partition to the communication partition during the synchronization phase. This doesn't incur to much overhead, as it is a copy operation from one cache location to another cache location. This time is small in comparison to the fetch time from main memory.

3.4 Architectural Requirements

The underlying architecture should have the following characteristics to allow a fast implementation of the proposed multithreading technique:

1. the processor provides the possibility to load data from main memory without bringing it into the cache, e.g. using cache hints,

2. multiple instructions can execute concurrently, e.g. a superscalar processor,

3. the processor does not stall on a cache miss, as long as independent instructions are available,

4. synchronization between threads is possible.

To our knowledge, no commercially available processor satisfies all above requirements. Some superscalar processors satisfy the first 2 conditions, but the third one is not met.

At first sight, it seems that out-of-order processors can continue executing independent instructions during a cache miss. However, they stall on main memory access[Pai *et al.*1999, Mowry *et al.*1998]. On current systems, a main memory access typically takes between 50 and 100 cycles. A current microprocessor is typically able to execute 4 instructions per cycle. To bridge the main memory access with useful computations, at least $50 \times 4 = 200$ independent instructions must follow the cache-missing instruction. The reservation stations in current processors can contain at most about 80 instructions. This results in a processor executing a maximum of about 80 independent instructions after the cache-missing instruction. When all these instructions are executed, the processor cannot get new instructions in the reservation station before the cache-missing instruction is completed because instructions must leave the reservation station in-order after finishing execution. When the processor can't find any more executable instructions in the reservation station, it stalls and waits until the requested data returns from main memory. In general, this limit is not a real problem, because only in a limited number of programs, 200 independent instructions following the memory access can be found. In this case, the limited instruction level parallelism would not allow faster execution with larger reservation stations. However, with cache remapping, when the communication thread accesses main memory, plenty of parallel instructions ready to execute can be found in the computation thread. To completely overlap the data fetch time, the computation thread must be able to continue computation on main memory access.

In a simultaneous multithreaded processor[Kwak *et al.*1999, Farcy *et al.*1996, Tsai *et al.*1999, Loikkanen *et al.*1996, Tullsen *et al.*1995] this problem does not arise since the cache-missing instructions reside in the data-fetch thread and the independent instructions reside in the computing thread. In multithreaded processors, instructions can leave the reservation station as soon as their execution is finished and all previous instructions *in the same thread* have finished. The computing thread can continue to fully use the computational resources of the processor while the data-fetch thread is waiting for the values returned from the main memory.

A processor allowing multiple non-blocking outstanding memory requests, and which is equipped with a pipelined memory subsystem, such as the Tera MTA memory system[Carter *et al.*1999], the execution time of the fetch thread can further be minimized by pipelining the data fetching.

4 Compiler Generated Data Fetch Thread

In the previous sections the framework of pipelined data-fetch and execution phases was laid out. In order to allow continuous computations, the data fetch phase should take less time than the parallel executing computation phase. This requires that the reuse and the locality of the data is optimized with respect to the limited cache footprints P_2 and P_3. Here one can use the results of a fertile area of data locality research, i.e. loop tiling.

4.1 Data Locality Transformation

Tiling is a well-known data locality improving transformation for array-based loop nests. It improves data locality by shortening the distances between consecutive reuses of array elements. It enables performing more useful computations in a smaller cache footprint. This is achieved by the following loop transformation.

Definition. Tiling[Wolf *et al.*1991] transforms an n-deep loop nest into a $2n$-deep loop nest. The *tiled* loops in the resulting loop nest are the n inner loops. The *tiling* loops are the n outermost loops. A loop nest will be denoted by $\mathcal{L}$. The tiled and the tiling loops for $\mathcal{L}$ are $\mathrm{Td}(\mathcal{L})$ and $\mathrm{Ti}(\mathcal{L})$ respectively. An *iteration tile* is the iteration space traversed by $\mathrm{Td}(\mathcal{L})$. The part of an array that is referenced during the execution of an iteration tile is a *data tile*. A *tile set* is the union of the data tiles of all the arrays accessed during an iteration tile execution. Tiling is legal when the loop nest is fully permutable.

The loop nest performing a matrix multiplication and the corresponding tiled loop nest is given in [Fig. 8].

Tiling eliminates many capacity misses, by making the long reference distances shorter. However, the low associativity of a cache may lead to a high number of conflict misses and slow down execution such that only a fraction of the attainable performance is obtained. Additional fine tuning of the tiling transformation is needed to reduce the conflict misses.

Cache remapping will eliminate the conflict misses by choosing a proper data placement in the cache. The tiling increases the reuse factor so that the cache remapping caching policy can work efficiently. The partitioning of the program is done as follows. Every execution of the tiled loops is mapped to one code partition. To make sure that condition (2) is satisfied, the tile sizes will be adapted.

4.2 Compiler Implementation

Profitability Analysis

The decision to apply the transformation to a loop nest can be made based on a prediction of the number and the kind of cache misses in the nest. The compile-

```
do i=1,N,1
  do j=1,N,1
    do k=1,N,1
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

(a) Original matrix multiplication

```
          do II = 1,N,Bi
  Ti(L)     do JJ = 1,N,Bj
              do KK = 1,N,Bk
L                 do i = II,min(II+Bi-1,N),1
        Td(L)       do j =  JJ,min(JJ+Bj-1,N),1
                      do k = KK,min(KK+Bk-1,N),1
                        C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

(b) Tiled matrix multiplication

Figure 8: Tiling applied to the matrix multiplication

time analysis of a loop nest's cache behavior has been described in [Ghosh1999, Harper *et al.*1999].

If the analysis predicts a lot of conflict misses, the threading transformation can improve execution speed. The first step is to tile the loop nest. Tiling a loop nest is simple, the transformation is described in [Section 4.1]. However, tiling should preserve the data dependencies of the original loop [Wolf1992].

Special Alias Resolution

To reduce the overhead of alias resolution during synchronization, described in [Section 3.3], the compiler techniques developed for scheduling independent tiles on multiprocessors can be used. For example, [Ramanujam *et al.*1992, Xue1997, Boulet *et al.*1994] all describe how to find independent tiles and how to schedule independent tiles to execute concurrently on different processors. The same techniques can be used to schedule the tiles on a uniprocessor such that two consecutive tiles are always independent. When this can be achieved, no data movement is necessary during the synchronization because no aliases arise.

Choosing Tile Sizes

In order to fit a maximal amount of data into a tile, the number of different memory elements addressed by one iteration tile $\mathrm{Td}(\mathcal{L})$ must be found. The iteration space of a tile is an integer polytope[Schrijver1986]. The number of iterations can be expressed as an Ehrhart polynomial[Clauss1996] with the tile sizes `Bi`, `Bj`, and `Bk` as parameters.

After generating a closed form expression for the number of iterations in the tiled loop and the number of data elements accessed during the execution of those iterations, the tile sizes are chosen such that:

– the total size of the accessed elements doesn't exceed the size of cache partition P_2 or P_3.

– the ratio r of the number of iterations to the number of elements that must be moved by the data-fetch thread is maximal.

The first choice ensures that every data tile set can completely reside in either cache partition P_2 or P_3. The data-fetch thread fetches the next data tile into one partition while the computing thread executes the current data tile in the other partition. The second constraint makes sure that the tile size will be chosen such that the reuse factor inside a partition is as large as possible.

Generating the Data-Fetch Thread

After the optimal tile size has been calculated, the data-fetch thread can be created. For the automatic creation of the data-fetch thread, the source code must meet the following requirements:

– The loop under consideration can be tiled.

– Every index in the array expression has the form $xi + c$, where x and c are loop-independent and i is an induction variable. x can be equal to 0.

– All references to the same array have the same index expressions, therefor all references to the same array access the same data region in an iteration tile. It is assumed that different arrays do not overlap, as is the case in Fortran.

The data-fetch thread must scan the data tile, and remap every element in it to a unique location in the cache shadow. The following procedure creates the data-fetch thread:

1. For every unique array reference `A(`$x_1 i_1 + c_1, \ldots, x_k i_k + c_k$`)`, a loop nest scanning the elements in the data tile is created. Let $I^{\mathtt{A}}$ be the vector containing the different induction variables in the reference. Then the statement

```
if (inside loop boundaries) then
  X(t) = A($x_1 i_1 + c_1, \ldots, x_k i_k + c_k$)
t=t+1
```

is embedded into a loop nest iterating over the values that the induction variables in $I^{\mathtt{A}}$ reach in the given tile. $D^{\mathtt{A}}$ is the diagonal matrix where element $D^{\mathtt{A}}_{ll}$ has the value $\prod_{j=1}^{l-1} \mathtt{B}_{I^{\mathtt{A}}_j}$, where $I^{\mathtt{A}}_j$ is the jth element of $I^{\mathtt{A}}$, and $\mathtt{B}_{I^{\mathtt{A}}_j}$ is the tile size of induction variable $I^{\mathtt{A}}_j$. $D^{\mathtt{A}}I^{\mathtt{A}}$ results in the value of `t` at iteration point $I^{\mathtt{A}}$ in the constructed loop.

   The variable `t` is initialized to zero after each synchronization, and `X` is the set to the start of either cache partition P_2 or P_3. The reference to `A` is annotated with the appropriate cache hints.

2. The loop nests created in this way are concatenated and synchronization is added. The code generated so far copies one data tile from the main memory to the cache shadow.

3. Put the generated code in the loop nest which iterates over the different tiles. In [Fig. 8], this is $\mathrm{Ti}(\mathcal{L})$.

Compile-time Address Relocation in the Computing Thread

The data-fetch thread moves array elements from its original locations to new locations in the cache shadow. Therefore, the references to the array elements in the computing thread must be substituted with references to the corresponding elements in the cache shadow. This is achieved by changing the references $\mathtt{A}(x_1 i_1 + c_1, \ldots, x_k i_k + c_k)$ into $\mathtt{X}(D^{\mathtt{A}}I^{\mathtt{A}})$.

Before executing a tiled loop nest, the computing thread synchronizes with the data-fetch thread to ensure that the next tile has completely been copied in the cache.

5 Evaluation

Between synchronization points, the number of iterations and data movements done by both threads is constant because the tile sizes are constant. Therefore, the exact amount of work between synchronization points is known and enables the interleaving and scheduling of both threads. This assumes no exceptional behavior like I-cache misses or exceptions. Because this technique works on tight scientific loop kernels, the static amount of code is small enough to fit in the instruction cache. Also, it is very uncommon for exceptions to occur inside these loop kernels.

No actual hardware implementation of an architecture meeting the requirements stated in [Section 3.4] is known to the authors. Therefore, the performance of the transformation was measured by simulating it on an explicitly parallel computing architecture (EPIC) which does meet the requirements. The execution speed of the multithreaded program is compared with the same program transformed by standard tiling algorithms.

5.1 Thread Scheduling

The instructions of the data-fetch thread are merged with the computation thread, so that the main memory accesses are equally spread over time. In this way the load on the memory subsystem to deliver data from main memory will be evenly divided over the complete computation.

The ratio r of main memory accesses to the number of computation iterations is introduced in [Section 4.2]. One main memory access is inserted after every r computation iterations, to evenly spread the main memory accesses.

The interleaving of iterations of both threads occurs at the source level. First, the loop nests from the data fetch thread are coalesced[Polychronopoulos1987] into single loops. Since the resulting loops are single loops, every iteration depends only on one iteration variable. The innermost loop of the computation thread is unrolled r times. After the unrolled loop body, the code for a single iteration from the data-fetch thread is inserted, after which the data-fetch loop index variable is updated. Every r computation iterations, one main memory access will be performed to update the cache shadow.

After this source level transformation, enough information about the independence of the instructions coming from the different threads is communicated to the *instruction scheduler*, to schedule the data-fetch thread instructions in parallel with the instructions of the computation thread. Also, the load and store instructions in the data-fetch thread which access memory locations outside the cache shadow are annotated to let the scheduler know that these will miss the cache and fetch from main memory. The scheduler takes this into account to hide the latency by issuing independent instructions from the computation thread during the memory fetch.

5.2 Results

Most capacity misses are resolved by the tiling transformation. The largest part of remaining cache misses are conflict misses. The data-fetch thread is able to resolve or hide the remaining misses, resulting in improved performance. The resulting performance is compared with other techniques to reduce the conflict misses[Lam *et al.*1991, Panda *et al.*1999, Rivera *et al.*1999, Temam *et al.*1993] in tiled algorithms. The conflict-reducing techniques do not hide the cold and left-over capacity misses. The only way to eliminate these misses is using some sort of prefetching. If prefetching is used without relocating the data, no improvements can be made[Saavedra *et al.*1996].

To get an idea of the effectiveness of threading the main memory accesses, we choose to simulate the execution of the matrix multiplication. This algorithm has become the de facto standard for comparing different tiling transformations. The original matrix multiplication and the different tiled versions are simulated

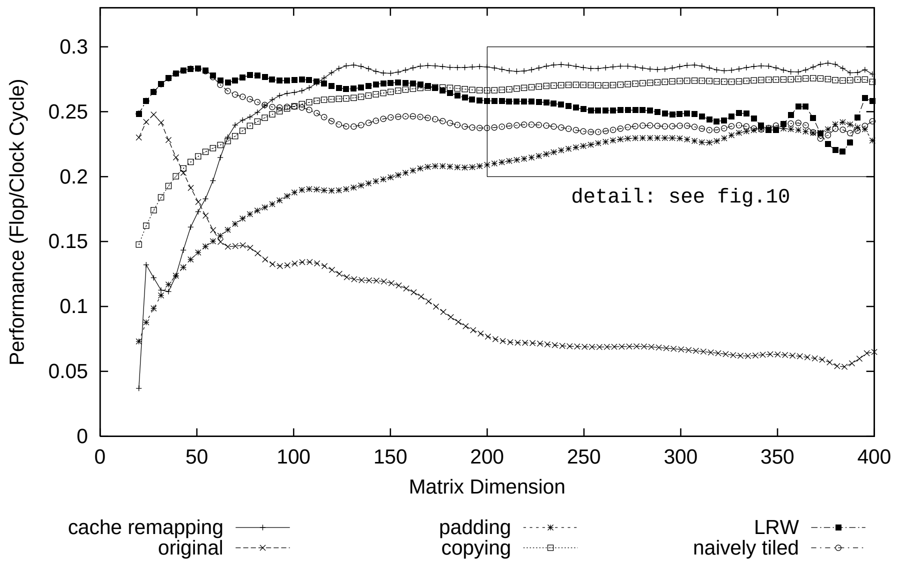


Figure 9: Smoothed plot of the performance of the tiled matrix multiplications for dimensions 20 to 400. This shows that cache remapping outperforms the alternative tiled algorithms for matrix size 150 and above. A zoom of the actual data can be found in [Fig. 10]

on a 8-issue EPIC processor. This also tests the effectiveness of compile-time thread scheduling. The Trimaran compiler and simulator[Trimaran1998] were used to optimize and execute the different versions of the matrix multiplication. The cache behavior was measured using the well known Dinero cache simulator[Hill *et al.*1989]. The following 2-level cache memory hierarchy was used. The L1 cache is 16Kb direct mapped with 32 byte lines. The L2 cache is 256Kb 4-way set associative with 64 byte lines. The access latency of the L2 cache is 20 clock cycles and the access latency of the main memory is 65 clock cycles.

The threaded version of the matrix multiplication is compared with the original program, a naively tiled version and three optimized versions corresponding to existing tiling algorithms[Panda *et al.*1999, Temam *et al.*1993, Lam *et al.*1991]. Each algorithm was coded, compiled and simulated for matrices with dimensions between 20 and 400.

In [Fig. 9], the average number of floating point operations per clock cycle is plotted. The plot is smoothed because of the irregularity in the data. The figure shows that cache remapping outperforms the alternative tiled algorithms from

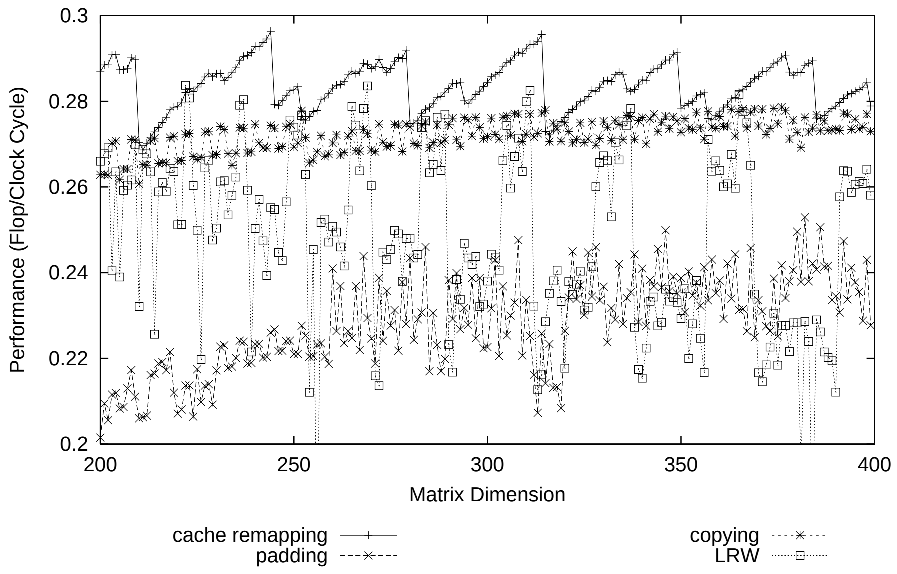


Figure 10: The performance of the cache remapping, padding[Panda *et al.*1999], copying[Temam *et al.*1993] and LRW[Lam *et al.*1991] on matrix dimensions 200 to 400. The threaded algorithm has the same performance as the next best algorithm at worst. At best, a speedup of 10% over the next best algorithm is obtained.

the point where the matrix dimension is big enough to hide the overlap of the pipeline start-up. In [Fig. 10], the actual data points are plotted for the range of matrix dimensions between 200 and 400. In this plot, we see that the threaded version outperforms the other versions.

At the border of the iteration space, the tiles are not completely filled with useful computation. The pipelined nature of cache remapping does not allow to process the less-filled tiles faster than completely filled tiles, because the processing of a less-filled tile in one thread is synchronized with the processing of a completely filled tile in the other thread. To alleviate this problem, the code was restructured to first process all complete tiles. After all completely filled tiles are processed, the less-filled tiles are processed. The amount of computations, data-fetch and the ratio r for these less-filled tiles vary with the matrix dimension. Therefore, the copying technique was used instead of cache remapping to process these tiles.

Variations in matrix dimension result in variations in the percentage of computation time needed to process the less-filled tiles. As the matrix dimension grows, the tiles at the borders become more and more filled with useful computations. Once the matrix dimension grows to big and a new set of tiles are needed to process the border, a new set of very sparse filled data tiles arise, and the performance drops a little. This explains the saw-toothed pattern observed in [Fig. 10].

The average speedup of the threaded code for matrix dimensions between 200 and 400 is 454% over the original code, 11% over the naively tiled code and 5% over the next best code, copying. The saw-tooth pattern leads to variations in the speedup over copying between 0 and 10%.

5.3 Related Work

Many optimizations have been proposed to ameliorate the memory behavior of programs (see introduction). Here, we'll only shortly discuss the techniques most closely related to ours.

Threaded processors have been proposed to hide the memory latency, by switching to another thread on a cache miss[Tullsen *et al.*1995, Mowry *et al.*1998], or by eliminating caches altogether and interleaving threads cycle-by-cycle to obtain a large amount of time between dependent instructions[Alverson *et al.*1995]. These multithreading techniques hide the memory latency, but do not ameliorate the cache use by minimizing conflicts, as cache remapping does. Because of the concurrent use of the cache by several threads, cache conflicts might even get worse because of the increased working set.

The combination of prefetch (to hide latency) and relocation (to resolve conflict misses) has been proposed by Yamada[Yamada *et al.*1994]. Instead of using an all-software multithreaded technique, Yamada proposes a hardware extension and new instructions allowing to fetch strided data from memory and place them in a single cache line. Loop unrolling and strip-mining are used as compiler transformations to overlap the prefetch latency with useful computations. In our work, we use the tiling transformation which allows a coarser grained synchronization between fetching and using the data. Basically, Yamada's technique allows to prefetch a constant number of cache lines, dependent on a small hardware buffer. Our technique allows to prefetch data tiles up to the half the cache size. This allows for a higher cache reuse when the algorithm processes $O(n^2)$ data by $O(n^3)$ computations, as is the case in many matrix computations.

The on-the-fly relocation in tiled algorithms has been described and studied in [Temam *et al.*1993, Wolf *et al.*1991]. However, during the data relocation no useful computations are performed. This leads to a lower overall performance. The other techniques to eliminate conflicts in tiled algorithms are choosing tile sizes without conflict misses[Lam *et al.*1991, Coleman *et al.*1995], which potentially results in small tiles and high loop overhead, and padding[Panda *et al.*1999] the complete array to obtain an array with a dimension which doesn't created many conflicts.

The idea to use multiple threads to improve the performance of single threaded programs has also been proposed at the micro-architectural level. Simultaneous subordinate microthreading[Chappell *et al.*1999] are threads written in microcode to support the execution of the main thread. The microthreads aim at improving the branch prediction, cache hit rate and prefetch effectiveness. The compiler is responsible for deciding when and which subordinate microthreads should be executed. The possibility to load the microcode of the subordinate threads into the processor makes it possible to implement multithreaded data-fetching on such a processor.

5.4 Future Work

The data-fetch thread resolves conflict misses and hides the latency of cold misses and capacity misses. If the number of capacity misses is too high, not enough computations can be found to hide the latency of the memory accesses. Tiling is one technique to reduce the number of capacity misses, but also other loop transformations can be used. Examples of such loop transformations can be found in [Yamada *et al.*1994, McKinley *et al.*1996]. Recently, some program transformations to shorten the reuse distance in irregular applications have been proposed[Mellor-Crummey *et al.*1999, Ding *et al.*1999]. It needs to be investigated how cache remapping can be used in these applications to remove the remaining conflict misses.

Software controlled cache placement gives rise to a number of interesting future research directions. As the compiler is able to control the caching behavior, it can incorporate full-program knowledge into the cache replacement algorithm as opposed to the online algorithms used in hardware. As knowledge of future references can be extracted at compile time, it would be very interesting to see how this could be used to implement a better replacement policy, e.g. based on the optimal algorithm of Belady[Belady1966], which indeed requires knowledge of future memory requests.

6 Conclusion

A technique is presented that allows a software controlled cache replacement policy on advanced processor architectures. This software control allows new advanced compiler optimizations, because the cache recently became visible to the software through cache hints. The proposed optimization improves the memory performance of single-threaded applications by spawning a second compiler generated thread which controls the caching of the data. Once the data-fetch thread is created, the program can run efficiently on a multithreaded processor.

Via explicit thread scheduling, it is possible to schedule both threads at compile time and run the resulting code on EPIC processors, where the dependence between instructions is explicitly coded in the instructions.

The proposed multithreaded technique can be exploited on any processor where parallelism can explicitly be communicated from the compiler to the processor. After measuring the execution time, we found that the proposed technique resulted in a speedup up to 10% compared with the next best memory optimization technique. Compared with the original code an average speedup of 450% has been obtained.

Acknowledgements

This research was supported by the Belgian government under contracts IWT-SB/991147 and GOA-12.0508.95.

References

[Alverson *et al.*1995] G. Alverson, S. Kahan, R. Korry, and C. McCann. Scheduling on the Tera MTA. *Lecture Notes in Computer Science*, 949, 1995.

[Baer *et al.*] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings, Supercomputing '91.*

[Belady1966] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[Boulet *et al.*1994] P. Boulet, A. Darte, T. Risset, and Y. Robert. (pen)-ultimate tiling? In IEEE, editor, *Proceedings of the Scalable High-Performance Computing Conference, May 23–25, 1994, Knoxville, Tennessee*, pages 568–576, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[Carter *et al.*1999] Larry Carter, John Feo, and Allan Snavely. Performance and programming experience on the Tera MTA. In *SIAM Conference on Parallel Processing*, March 1999.

[Chappell *et al.*1999] Robert Chappell, Jared Stark, Sangwook Kim, Steven Reinhardt, and Yale Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the The 26th International Symposium on Computer Architecture*, may 1999.

[Clauss1996] Philippe Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*. ACM, May 1996.

[Coffman *et al.*1973] E. G. Coffman and P. J. Denning. *Operating Systems Theory.* Prentice Hall, Englewood Cliffs, 1973.

[Coleman *et al.*1995] Stephanie Coleman and Kathryn McKinley. Tile size selection using cache organization and data layout. In *SIGPLAN'95: conference on programming language design and implementation*, pages 279–290, June 1995.

[Ding *et al.*1999] Chen Ding and Ken Kennedy. Improving cache performance of dynamic applications through data and computation reorganization at run time. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, pages 229–241, May 1999.

[Farcy *et al.*1996] A. Farcy and O. Temam. Improving single-process performance with multithreaded processors. In *FCRC '96: Conference proceedings of the 1996 International Conference on Supercomputing*, pages 350–357, 1996.

[Ghosh1999] Somnath Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behaviour.* PhD thesis, Princeton University, November 1999.

[Harper *et al.*1999] John S. Harper, Darren J. Kerbyson, and Graham R. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transaction on Computers*, 48(10):1009–1024, oct 1999.

[Hill *et al.*1989] Mark D. Hill and Alan Jay Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[IA64 ADAG1999] *IA-64 Application Developer's Architecture Guide*, May 1999.

[Irigoin *et al.*1988] F. Irigoin and R. Triolet. Supernode partitioning. In *POPL '88. Proceedings of the conference on Principles of programming languages*, pages 319–329, 1988.

[Jouppi1990] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th ISCA*, pages 364–373, May 1990.

[Kane1996] Gerry Kane. *PA-RISC 2.0 architecture.* Prentice Hall, 1996.

[Kathail *et al.*2000] Vinod Kathail, Michael S. Schlansker, and B. Ramakrishna Rau. HPL_PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard, February 2000.

[Kwak *et al.*1999] Hantak Kwak, Ben Lee, Ali R. Hurson, Suk-Han Yoon, and Woo-Jong Hahn. Effects of multithreading on cache performance. *IEEE Transactions on Computers*, 48(2):176–184, February 1999.

[Lam *et al.*1991] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[Loikkanen *et al.*1996] Mat Loikkanen and Nader Bagherzadeh. A fine-grain multithreading superscalar architecture. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 163–168, October 20–23, 1996.

[McKinley *et al.*1996] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[Mellor-Crummey *et al.*1999] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 1999 Conference on Supercomputing*, pages 425–433, June 20–25 1999.

[Mowry *et al.*1992] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. *ACM SIGPLAN Notices*, 27(9):62–73, September 1992.

[Mowry *et al.*1998] Todd C. Mowry and Sherwyn R. Ramkissoon. Software-controlled multithreading using informing memory operations. Technical report, School of Computer Science - Carnegie Mellon University, 1998.

[Mowry1994] T. Mowry. *Tolerating Latency Through Software Controlled Data Prefetching.* PhD thesis, Dept. of Computer Science, Stanford University, March 1994.

[Pai *et al.*1999] Vijay S. Pai and Sarita Adve. Code transformations to improve memory parallelism. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 147–155, november 1999.

[Panda *et al.*1999] P. Panda, H. Nakamura, N. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE transactions on computers*, 48(2):142–149, Feb 1999.

[Polychronopoulos1987] C. D. Polychronopoulos. Loop coalesing: A compiler transformation for parallel machines. In *International Conference on Parallel Processing*, pages 235–242, August 1987.

[Pyo *et al.*1997] Changwoo Pyo and Gyongho Lee. Reference distance as a metric for data locality. In *HPC-ASIA 97*, pages 151–156, 1997.

[Ramanujam *et al.*1992] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, October 1992.

[Rivera *et al.*1998] Gabriel Rivera and Chau-Wen Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 38–49, June 1998.

[Rivera *et al.*1999] G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th International Conference on Compiler Construction (CC'99)*, March 1999.

[Saavedra *et al.*1996] R.H. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *10th Int. Parallel Processing Symp. (IPPS'96),*, april 1996.

[Schrijver1986] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, New York, 1986.

[Seznec *et al.*1993] André Seznec and Francois Bodin. Skewed-associative caches. In *Proceedings of PARLE '93 – Parallel Architectures and Languages Europe*, pages 305–316, Munich, Germany, June 14–17, 1993.

[Temam *et al.*1993] Olivier Temam, Elana D. Granston, and William Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings, Supercomputing '93*, pages 410–419, March 1993.

[Trimaran1998] Trimaran. *The Trimaran Compiler Research Infrastructure for Instruction Level Parallelism.* The Trimaran Consortium, 1998. http://www.trimaran.org.

[Tsai *et al.*1999] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, sept 1999.

[Tullsen *et al.*1995] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, June 1995.

[Weber *et al.*1989] Wolf-Dietrich Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, May 1989.

[Wolf *et al.*1991] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, 1991.

[Wolf1992] M.E. Wolf. Improving locality and parallelism in nested loops. Ph.d. thesis, Stanford University, 1992.

[Xue1997] Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[Yamada *et al.*1994] Yoji Yamada, John Gyllenhaal, Grant Haab, and Wen mei Hwu. Data relocation and prefetching for programs with large data sets. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 118–127, November 30–December 2, 1994.