Cache remapping to improve the performance of tiled algorithms

Kristof Beyls and Erik D'Hollander

University of Ghent
Department of Electrical Engineering
St.-Pietersnieuwstraat 41
B-9000 Gent, Belgium

Abstract. With the increasing processing power, the latency of the memory hierarchy becomes the stumbling block of many modern computer architectures. In order to speed-up the calculations, different forms of tiling are used to keep data at the fastest cache level. However, conflict misses cannot easily be avoided using the current techniques. In this paper cache remapping is presented as a new way to eliminate conflict as well as capacity and cold misses in regular array computations. The method uses advanced cache hints which can be exploited at compile time. The results on a set of typical examples are very favorable and it is shown that cache remapping is amenable to an efficient compiler implementation.

1 Introduction

With Moore's Law still doubling the performance in 18 months, there is almost no limit to the processing power for the foreseeable future. Many performance programmers know that this is not the case, due to the speed gap between the processor and the memory. In fact, where the processor speed gains about 67% per year, the memory lags behind with only a gain of about 5-10%[3]. Using the same reasoning as Moore, one quickly finds out that a similar law says *the memory speed with respect to the processor halves each 22 months...* . From this observation, the growing importance of L1, L2 and L3 caches becomes evident and the objective is to keep the used data in the cache all the time.

Tiling[1, 15] is a well known method to improve the reuse of cached data in numerical applications by shortening the distance between the use and reuse of array elements. Tiling algorithms successfully eliminate capacity misses and therefore increase the cache hit ratio. However, the low associativity of caches may lead to a high number of conflict misses and slow down execution so that only a fraction of the attainable performance is obtained. Additional fine tuning of the tiling transformation is needed to reduce the conflict misses[2, 7, 9, 11, 13].

Research financed by the Flemish government under contracts IWT-SB/991147 and GOA-12.0508.95)

In this paper, cache remapping is offered as a new technique to eliminate conflict misses in tiled algorithms. In addition, cache remapping produces no capacity misses and also cold misses are avoided for all but the first iteration.

Cache remapping is based on a dynamic rearrangement of the data at run time. During the execution of a loop, a parallel remap thread running concurrently with the original program thread relocates the tiled data needed by future iterations. The cache is split in two regions. One region contains all the data in the tile currently being processed, enabling the calculations to continue without memory stalls. At the same time the remap thread copies the data of the next tile into the other cache region, using proper address relocation and cache bypass. When the calculations have completed processing a tile, the original processing thread can immediately continue processing the next tile as it is already brought in the cache by the remap thread.

Section 2 explains cache remapping in detail. In section 3, experimental results are presented. Section 4 compares the techniques and the results with related work.

2 Cache remapping technique

2.1 Tiled Loop Nests

Fig. 1 shows a loop nest and the corresponding tiled loop.

Definition 1. *Tiling[15] transforms an n-deep loop nest into a 2n-deep loop nest. The* tiled *loops in the resulting tiled loop nest are the n inner loops. The* tiling *loops are the n outermost loops. A loop nest will be notated as $\mathcal{L}$. The tiled and the tiling loops for $\mathcal{L}$ are $\mathcal{T}d(\mathcal{L})$ and $\mathcal{T}i(\mathcal{L})$ respectively. An* iteration tile *is the iteration space traversed by $\mathcal{T}d(\mathcal{L})$. 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.*

```
do i=1,N,1
  do j=1,N,1
    do k=1,N,1
      H(i,j,k)
```

(a) Original loop nest

$\mathcal{L}$ ⎛ $\mathcal{T}i(\mathcal{L})$ ⎛
```
do II = 1,N,B1
  do JJ = 1,N,B2
    do KK = 1,N,B3
```
$\mathcal{T}d(\mathcal{L})$ ⎛
```
do i = II,min(II+B1-1,N),1
  do j = JJ,min(JJ+B2-1,N),1
    do k = KK,min(KK+B3-1,N),1
      H(i,j,k)
```

(b) Tiled loop nest

Fig. 1. A tiled loop nest

2.2 Cache Memory

For the development of the cache remapping technique, a cache is represented by a tuple (C_s, N_s, k, L_s)[4]

Definition 2. *The* cache size (C_s) *defines the total number of bytes in the cache. The* line size (L_s) *determines how many contiguous bytes are fetched from memory on a cache miss. A* memory line *refers to a cache-line-sized block in the memory which is aligned such that the data in it map into the same cache line. A* cache set *is the collection of cache lines in which a particular memory line can reside. 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 $C_s = N_s \times k \times L_s$.*

The start address A of a memory line determines the cache set N it maps to:

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

A replacement algorithm decides the cache line in set N a memory line is copied to. In the rest of this paper the least recently used (LRU) replacement policy is assumed.

Definition 3. *Consider the memory lines accessed during the execution of $\mathcal{L}$. Then* $\mathrm{Ml}(\mathcal{L}, N)$ *represents the set of memory lines which map to cache set N.*

2.3 Conflict Misses in Tiled Algorithms

Consider a tiled loop $\mathrm{Td}(\mathcal{L})$ and a cache set N. When $\#\mathrm{Ml}(\mathrm{Td}(\mathcal{L}), N) > k$, more than k memory lines must be placed in the same cache set N. Only part of the memory lines can reside in the cache at the same time, and conflict misses arise.

Cache remapping copies tile sets into a contiguous C_s-sized buffer. Because of (1), k memory lines in the buffer map to each cache set. So, $\forall N \in N_s$: $\#\mathrm{Ml}(\mathrm{Td}(\mathcal{L}), N) = k$ and no conflict misses arise.

2.4 High-Level View of Cache Remapping

Cache remapping adds a remap thread to the program, which executes concurrently with the original processing thread executing the tiled loop nest $\mathcal{L}$ (see fig. 2). These two threads can execute in parallel on processors with multiple functional units. (for further detail, see sect. 2.5).

Consider an iteration point i of $\mathrm{Ti}(\mathcal{L})$. The two threads work in a pipelined way(see fig. 3):

- The *processing thread* executes tile i.
- The *remap thread* copies data tile set $i+1$ into the cache. If there is written data of tile set $i-1$ in the cache, it is first copied back to main memory to make place for tile set $i+1$.

At most two consecutive tile sets are in the cache at the same time. Between iterations of $\mathrm{Ti}(\mathcal{L})$, the two threads synchronize.

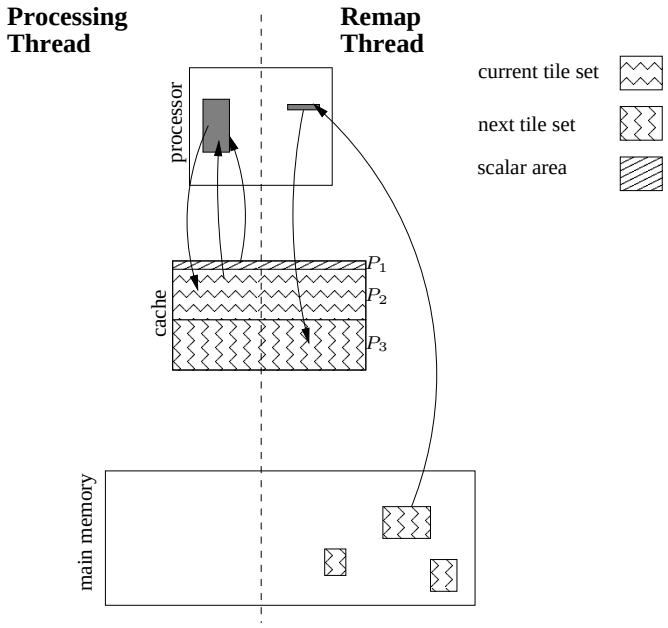

Fig. 2. The remap thread puts the next tile set in the cache while the original thread processes the current tile set. In the next phase, the processing and the remap thread will access P_3 and P_2 respectively.

Fitting the Current and Next Tile Set in the Cache The process thread accesses two kinds of variables in the memory: scalar variables which do not fit into the registers and arrays. To ensure that all data referenced by the process thread is in the cache, it is logically divided in three partitions: P_1, P_2 and P_3. P_1 is used to cache the scalars. P_2 and P_3 will each contain one tile set.

During the odd iterations of $\mathrm{Ti}(\mathcal{L})$, the process thread uses P_2, during the even iterations, it accesses P_3. The remap thread uses P_3 during the odd iterations and P_2 during the even iterations. It is clear that P_2 and P_3 must have the same size as they are used symmetrically.

Respecting Data Dependencies Problems arise when there are data dependencies between the tile sets of two consecutive iteration tiles.

If the process thread currently processes tile set i and writes into elements of tile set $i+1$, the remap thread prefetches these elements into the cache with the old values. When the process thread executes tile $i+1$, it will use the old values instead of the new.

A solution is to copy the new value of the shared elements to the cache partition the process thread will use. This must be done during the thread synchronization, which occurs between iterations of $\mathrm{Ti}(\mathcal{L})$.

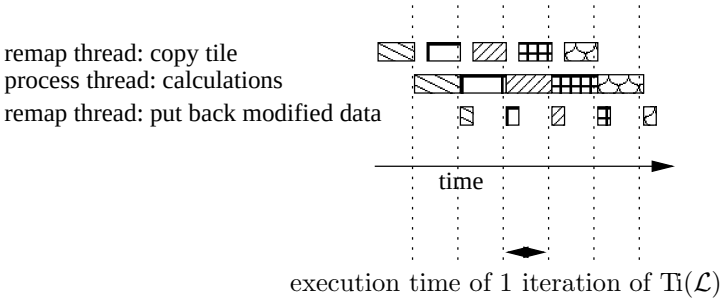

Fig. 3. The pipelined nature of cache remapping

2.5 Low-level Details

Controlling Cache Behavior

Cache Shadow At the start of the program, a consecutive block of memory with size C_s is allocated and aligned on a memory line. We call this memory block the cache shadow. There's a one-to-one relation between the addresses in the cache shadow and the storage area in the cache. The area's P_1, P_2 and P_3 are allocated in this cache shadow.

To assure that the contents of the cache shadow always resides in the cache completely, cache hints are used. They make it possible to only cache the addresses in the cache shadow by bypassing the cache on memory references outside the cache shadow.

Cache Hints In modern instruction sets, cache hints[5, 6] are attached to load and store instructions. They specify if the referred data should be cached or not. When data is loaded/stored from/to P_1, P_2 or P_3, a cache hint tells the processor to cache the data. If an address outside the cache shadow is referenced, the cache hint tells the processor not to cache the data.

Thread Scheduling on a Single Threaded Superscalar Processor The process and remap threads need to run concurrently. Current microprocessors offer parallelism at the instruction level (ILP). This means that only nearby instructions in one thread can be executed simultaneously. To execute the remap thread and the process thread concurrently, these two threads need to be interwoven into a single thread at compile time. The instructions of the two threads must be interleaved so that the processor can execute instructions of the two threads during the same cycle.

On current processors, about a dozen functional units are present. Typically, data dependencies cause an average IPC (instructions per clock-cycle) no more than 2 to 3, so about ten functional units are left unused every clock-cycle. There are no dependencies between the remap and the process thread during the execution of Td($\mathcal{L}$). As a result, the remap thread can use the functional

units that are not used by the process thread. A good optimizing compiler can schedule the instructions of both threads so that they execute simultaneously.

Non-Stalling Memory Access The remap thread accesses main memory. Because the two threads are interwoven into one, it is important that the memory access doesn't stall the processor. When an instruction from the remap thread accesses main memory, there are enough independent instructions from the process thread ahead in the instruction stream to perform useful in-cache computations to overlap the latency.

Selection of Tile Size The tile size (B_1,...,B_n; $n = 3$ in the example) is chosen so that every tile set fits in P_2 and P_3. A large number of tile sizes satisfy this constraint. Let $iter_p = B_1 * \cdots * B_n$, the number of iterations executed by the process thread during a tile execution. Let $iter_r$ be the number of array elements that need to be remapped or put back during a tile execution. We choose to optimize the tile sizes so that the ratio $r = \frac{iter_p}{iter_r}$ is maximal. This choice assures that the processing power needed by the remap thread is as small as possible relative to the processing power needed by the process thread.

Loop Transformations and Thread Scheduling To lower the scheduling overhead, a number of loop transformations are performed to the loop nests in both threads. The remap thread originally consists of Q loop nests. Every loop nest remaps or puts back a data tile. Q_i^r is the number of elements that are remapped by loop nest i. Each of these loop nests are coalesced[10], and the body of the remaining loop is placed in an inlined remapping function (e.g. `remapA` in fig. 4(a)).

The innermost loop in the tiled loop nest $\mathrm{Td}(\mathcal{L})$ is unrolled $\lfloor r \rfloor$ times, then a remap call is inserted (see fig. 4(c)).

It is known at compile time how many times each remap function must be executed per iteration of $\mathrm{Td}(\mathcal{L})$. The outermost loop of $\mathrm{Td}(\mathcal{L})$ is split into Q parts. In each part, another remap function is called. The number of iterations in each part is chosen so that every remapping function is called at least Q_i^r times. So $Q_i^{B_1} * B2 * \left\lfloor \frac{B3}{r} \right\rfloor \geq Q_i^r$.

3 Implementation and results

3.1 Processor Requirements

Three conditions must be met to enable cache remapping:

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 execute concurrently, e.g. a superscalar processor,
3. the processor does not stall on a cache miss, as long as independent instructions are available in the instruction stream. This can be achieved using speculative loads[5].

The IA-64 architecture satisfies these requirements as well as all Explicitly Parallel Instruction Computing (EPIC)-style architectures.

```
remapA(int iter,A,p) {
  i1 = de_coalesce(iter);
  i2 = de_coalesce(iter);
  remap(p+i1*B2+i2,
        A[i1+II,i2+JJ]);
}
```

(a) One of the Q functions that remap one element

```
remap(double* x, double* y)
{
   fld.nta  r1,y
   fst.nt1  x,r1
}
```

(b) The remap function. The `nta` cache hint means "don't cache", the `nt1` cache hint means "cache into L1".

```
swap(p2,p3)
iter=0
do i = II,II+Q1^B1-1
  do j = JJ,JJ+B2-1
    do k = KK,KK+B3-1,r
      H(i,j,k,p2) /* body r */
      ... /* times unrolled */
      H(i,j,k+r-1,p2)
          /* remap code */
      remapA(iter++,A,p3)
iter=0
do i = II+Q1^B1,II+Q1^B1 + Q2^B1-1
  do j = JJ,JJ+B2-1
    do k = KK,KK+B3-1,r
      ...
      remapB(iter++,B,p3)
...
```

(c) The transformed tiled loop nest.

Fig. 4. The program transformations to efficiently interweave and schedule both threads into one. `p2` and `p3` are the start addresses of P_2 and P_3 respectively. It is assumed that — after inlining — the instruction scheduler will move enough independent instructions between both instructions in `remap` to allow useful computations during the main memory access.

3.2 Simulation

Since EPIC processors are not yet available, the Trimaran simulator[14] was used to simulate the behavior of the processor. The cache behavior was modeled by the well known Dinero cache simulator.

The experiment is a tiled matrix multiply executed on a processor with a 2-level cache. The L1 cache is 16Kb direct mapped with 32 byte lines. The L2 cache is a 256Kb 4-way set associative with 64 byte lines. We assume that the access latency of the L2 cache is 20 clock cycles and the access latency of the main memory is 65 clock cycles.

The cache remapping technique was compared with the original algorithm, a naively tiled algorithm not considering limited cache associativity and three optimized tiling algorithms, namely padding[9], copying[13] and LRW[7]. Each algorithm was coded, compiled and simulated for matrix dimensions between 20 and 400. For the cache remapping algorithm, the tiles on the border of the iteration space were processed using the copying technique, because the pipelined

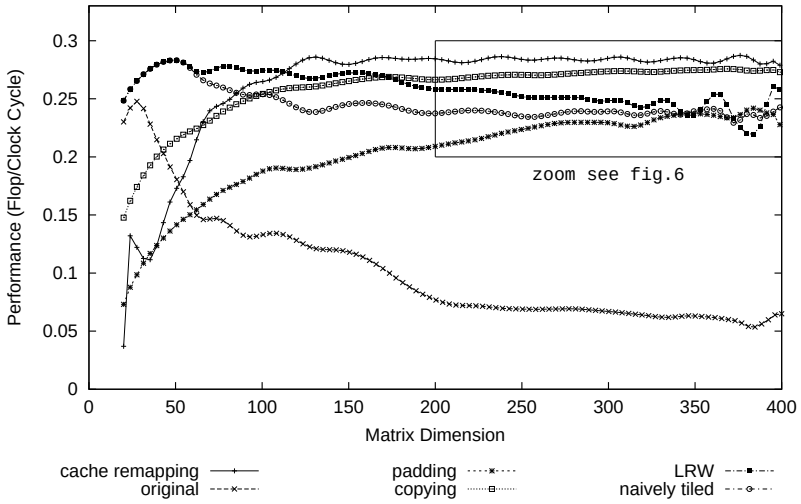

Fig. 5. Smoothed plot of the performance of several tiled matrix multiplications for dimensions 20 to 400. In this smoothed plot it is clear that the cache remapped algorithm outperforms the others for matrix sizes bigger than 150. A zoom of the actual performance plot can be found in figure 6.

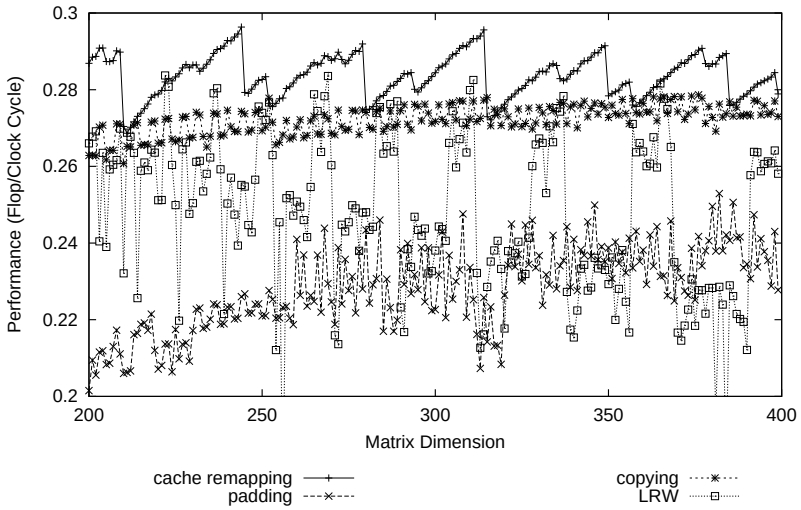

Fig. 6. The performance of cache remapping, padding[9], copying[13] and LRW[7] on matrix dimensions 200 to 400. The cache remapped algorithm has the same performance as the next best algorithm at worst. At best, a speedup of 10% over the next best algorithm is obtained.

nature of cache remapping suffers from processing tiles not completely filled with data.

The performance of the algorithms, expressed in number of floating point operations per clock cycle, is plotted in figure 5. Because the performance of some algorithms fluctuate, the data was smoothed using bezier curves to clearly visualize the trends. In figure 6 an exact plot is given for the four best algorithms for matrix dimensions 200 to 400. This plot shows that at worst, cache remapping is as good as, and at best it is 10% better than the next best algorithm.

For matrix dimensions bigger than 150, cache remapping outperforms the alternative tiled algorithms. For matrix dimensions between 200 and 400, the average speedup compared to the second best algorithm (copying) is 5%. Compared with the original non-tiled algorithm, a speedup of 454% is obtained.

4 Comparison with related work

Methods that select tile sizes to eliminate conflict misses[2, 7] sometimes result in small tiles, which reduce the performance. Padding[9] on the other hand uses large tile sizes and changes the data layout of the arrays by enlarging the dimensions with unused elements, in order to avoid cache conflicts. Unfortunately, this static adjustment cannot be optimized for every loop in a program simultaneously. Copying[13, 7, 11] eliminates conflict misses by copying the array tiles with the worst self interference to a contiguous buffer. Copying naturally involves overhead and the tradeoffs between copying and cache conflicts are discussed in [13].

In contrast to Padding and Tile Size Selection, cache remapping is independent of the array dimensions and doesn't require a change of the data layout. With respect to copying, cache remapping is able to cache tiles in a parallel thread, which runs concurrently with the processing thread. As a consequence, cache remapping has no conflict misses and incurs a minimal overhead.

The cache bypass and relocation technique was exploited by Lee[8] to use the cache as a set of vector register on i860 processors mimicing Cray's strided get/put[12]. Yamada[16] proposed prefetching and relocation by extending the hardware with a special data fetch unit which enables prefetching strided data without cache pollution. Our technique also combines cache bypass and relocation, but isn't limited to strided data patterns which allows it to prefetch and relocate data structures with non-constant strides such as data tiles.

5 Conclusion

The Von Neumann bottleneck nowadays hinders even a single processor. Cache remapping represents a promising technique to bridge the steadily growing gap between processor and memory speeds. It favorably compares with existing tiling techniques and it uses the concepts of a new generation of processors. In future work the presented technique will be embedded in a EPIC compiler.

References

1. S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings, Supercomputing '92*, pages 114–124. IEEE Computer Society Press, November 1992.
2. S. Coleman and K. 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.
3. D. P. et al. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, March-April 1997.
4. S. Ghosh. *Cache Miss Equations: Compiler Analysis Framework for Tuning Memory Behaviour*. PhD thesis, Princeton University, November 1999.
5. *IA-64 Application Developer's Architecture Guide*, May 1999.
6. G. Kane. *PA-RISC 2.0 architecture*. Prentice Hall, 1996.
7. 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, Palo Alto, California*, pages 63–74, April 1991.
8. K. Lee. The NAS860 library user's manual, 1993.
9. 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.
10. C. D. Polychronopoulos. Loop coalesing: A compiler transformation for parallel machines. In *International Conference on Parallel Processing*, pages 235–242, Pennsylvania, Pa, USA, Aug. 1987. Pennsylvania State Univ. Press.
11. G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th International Conference on Compiler Construction (CC'99)*, March 1999.
12. S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. ASPLOS VII*, Cambridge, MA, Octobe 1996.
13. O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In IEEE, editor, *Proceedings, Supercomputing '93*, pages 410–419, March 1993.
14. Trimaran. *The Trimaran Compiler Research Infrastructure for Instruction Level Parallelism*. The Trimaran Consortium, 1998. http://www.trimaran.org.
15. 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.
16. Y. Yamada, J. Gyllenhaal, G. Haab, and W. 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, San Jose, California, Nov. 30–Dec. 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.