Block Prefetching for Numerical Codes

Josef Weidendorfer and Carsten Trinitis*
Technische Universität München
{weidendo,trinitic}@cs.tum.edu

Abstract

Cache optimization is a crucial technique for most numerical code to exploit the performance of modern processors. It can be classified into improving access locality, and prefetching. Inherent algorithm constrains often limit the first approach which typically uses a blocking technique. While there exist automatic prefetching mechanism in hardware and/or compilers, they can not complement blocking with additional prefetching.

We describe application controlled block prefetching, allowing to further improve on numerical code already optimized by blocking. It shows its benefits on both synthetic code and matrix multiplication by using the 2nd core of a dual-core processor as engine for executing block prefetching instructions.

1 Introduction

Because of the growing gap between main memory and CPU performance, cache optimization is an important technique for numerous numerical codes to exploit the performance of modern processors. A standard strategy tries to improve the locality of memory accesses, i.e. it improves the probability that on a data access, the data already resides in the cache. However, sometimes algorithms inherently limit the locality reachable this way; the result is that the computational performance is still way below the processor's theoretical capacities. A complementing cache optimization technique is *prefetching*: it tries to load required data into cache previous to the actual access. However, prefetching can evict other data which should be available for reuse; by neglecting this possibility, application performance could even get worse. Therefore, a rule is to load the required data as close as possible before its actual use. This type of prefetching often can be done automatically in hardware, e.g. by detecting stream accesses in the cache controller [DDS95] or by speculatively pre-executing the code using run-ahead execution [MKP05]. The latter also can be done in software by a *helper thread* [KLea04] automatically generated by the compiler. Via source code modification, compilers allow the use of prefetch directives: this inserts software prefetch instructions available on most processors, that fetch a cache line containing a given address into the processor.

The before-mentioned directly-before-use prefetching has one disadvantage: it is useless if executed in a code region which already uses the full bandwidth to main memory; prefetching can not push this limit further. For this kind of code, long-distance prefetching is required: if there is a program phase with low or no bandwidth demand to main memory,

*Department of Informatics, Technische Universität München, D-85747 Garching bei München, Germany

prefetching the data for a subsequent program phase with high bandwidth requirements is beneficial. The data to prefetch typically consists of blocks of data. Therefore, this long-distance prefetching is called *block prefetching*.
In this paper, we first give some related work. In chapter 3, the basic infrastructure is given for off-loading prefetching on a second core. Afterwards, we present some applications and measurements to check for benefits of this structure. The paper finishes with some conclusions and future work issues.

2 Related work

When there is no extra support for block prefetching in a single-threaded processor, one has to interleave prefetching with the regular application code. In [WT06], we have shown that in simple cases this is possible. But even with modest complex code, the additionally required code using deeper loop nests makes this approach impossible because of added instruction overhead and limits on the instruction decoding bandwidth. Yet, [Bey04] shows that this technique is beneficial on Itanium processors; Beyls reorganizes data layout while prefetching into cache, thus avoiding possible conflict misses in the cache via *cache remapping*. As we will see, our off-loading allows to re-layout in a quite effective way.
With multi-threading, prefetching can be done in another simultaneous running thread: however, our experiments with *Hyperthreading* on Intel processors showed problems because a running hyperthread slows down the other thread to 70 percent of its original performance, thus making it difficult to get benefits. However, other work has shown that is is possible to achieve speedups this way [KLea04].
Prefetching by using copy operations allows to rearrange data in the cache in such a way that accesses to the copy destination minimizes conflict misses in contrast to the original data. This technique, which is possible with our infrastructure, is also analyzed in [BS97], where it is compared to a hardware approach called *skewed associativity* to minimize conflict misses.

3 Block Prefetching Infrastructure

The best way to do block prefetching would be to use built-in processor instructions which prefetch a given range of addresses into cache. We do not know about processors with such instructions. However, there are multi-core processors where cores share a cache. In our experiments, we use the Intel Core Duo (Yonah) and Intel Core 2 Duo (Woodcrest) processors, which both have two cores sharing the L2 cache.
Figure 1 shows the structure. The prefetch infrastructure actually is implemented as a library[1], as given in table 1.
First, the implementation checks whether it is really running on a multi-core processor. Spawning a second thread on a uni-core would be counterproductive; in this case, doing block prefetching in a synchronous way in the main application can still be beneficial.

[1] The code will be available on `http://mmi.in.tum.de`

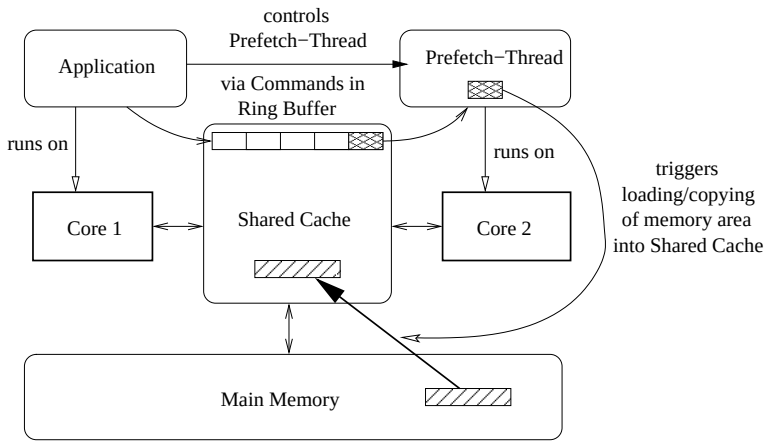

Figure 1: Infrastructure for off-loading prefetching to another processor core

If we are running on multi-core, the helper thread is started. The framework uses a ring buffer communication to off-load prefetch/copy requests from the main application to the prefetch thread; it does busy waiting on the ring buffer looking for commands from the main application. We note that from a power efficiency point of view, busy waiting is suboptimal; however, we need the fast synchronization.

We have to make sure that the prefetch thread is running on a core nearside the core which runs the main application, so that both cores share a cache. In a multiprocessor machine on Linux, this can be achieved by setting a threads CPU affinity mask to bind it to a given processor/core, thus forcing the needed neighborhood.

A basic command for the prefetch thread is to access a memory block sequentially, thus loading the data into the shared cache. It is enough to access each cache line only once, which maps to an access every 64 bytes on the Intel CPUs. However, accesses from both cores to the shared cache itself can create a bottleneck: theoretically, the prefetch thread can slow down the main thread this way by hamper main application accesses. A solution is to throttle the prefetch thread as needed by artifically introducing some dummy working load while the prefetch command is executed. However, in the case of the Intel CPUs, we did not notice any benefit in throttling the prefetch thread. Some more detailed analysis is worked on.

Table 1 mentions one advanced prefetch request, which actually consists of multiple copy operations. It is used to copy regularly scattered memory blocks into one sequential larger block. Copying may appear counterproductive as prefetch operation; however, it makes sure the data is available in the cache at the copy destination. As long as this destination is overwritten by a further prefetch command before these cache lines are evicted from the cache, no write transaction will appear for the modified cache lines at the copy destination when a write-back cache is used. This means that such copy commands will not generate more traffic on the memory bus. The copy operations allow data from memory to be rearranged in the cache for more compact and therefore faster access. Additionally,

Command	Comment
`pref_init()`	Initialize prefetch thread
`pref_fetchBlock(void* addr, int size)`	Trigger block prefetching
`pref_joinBlocks(void* dst, void *src, int blen, int bstride, int blocks)`	Trigger joining copy of blocks
`pref_fini()`	Terminate prefetch thread

Table 1: API of prefetch library

```
/* size % block == 0 */
double* a[size];
for(i=0;i<size;i+=block)
  if (i>0 && i<size-block)
    fetchBlock(a+i+block,block)
  for(iter=0;iter<N;iter++)
    for(j=0;j<block;j++)
      sum += a[i+j]
```

Prefetch Version	N=2	N=3	N=4
	[MFlop/s]		
Without	675	789	852
Interleaved	683	697	706
Pf. Thread	940	1003	1038

Figure 2: Synthetic one-dimensional blocking benchmark

simultaneous access of the original spatially separated memory blocks can produce a high number of conflict misses, depending on the cache associativity. This problem is minimized by accessing the large, sequential block instead. Furthermore, hardware prefetch schemes working at the L1/L2 border benefit from sequential access.
Prefetching by copying pollutes double the space of the data when the copy is done. Therefore, the later access benefits have to compensate this effect to be useful. However, when the processor provides load instructions directly from main memory into registers, this can be used in the copy operation, making it as efficiently as a pure block prefetch. The Itanium has such an instruction; the Intel IA-32 processors provide *non-temporal* loads, which pollute only one way in a multi-way associative cache, thus reducing the mentioned effect.

4 Results

First, we run a synthetic vector sum benchmark using one-dimensional blocking: while the main thread sweeps a few times (N) over a memory block, the next block is loaded into cache. Similar code was already used in [WT06]; there, we interleaved the prefetching into the main thread, complicating the main application thread by introducing further nested loops. With our new framework, the main code is modified only by the insertion of the function requesting the prefetch for the next memory block. Figure 2 shows the benchmark code with the request to the prefetch thread, and the measurement results for the code without any prefetching, with the old interleaved prefetching (using software prefech instructions), and with prefetching via our new infrastructure (as given on the left).
Measurement were done on a 2.16 GHz Intel Core Duo (Yonah) processor, with `size=`

MM Version	Performance [GFlop/s] Block=4	Block=8	Block=50
Block Size [KB]	32	64	400
Naive	1.02	1.02	1.02
Blocked	1.95	2.38	2.90
Bl.+Pf.	2.26	2.52	2.84

Figure 3: Matrix Multiplication performance on Woodcrest

1M, `block` = 16K, and compilation flags `-O3 -xP`. While on the Pentium-M, which was used in the original paper on interleaved prefetching, there was quite some benefit, on the newer processor there are negative effects because of the instruction overhead of the additional nested loops. With the prefetching thread, we can see major benefits in all measured cases.

The second application we did measurements with is the matrix multiplication: with appropriate 2D-blocking, e.g. reusing a matrix block 50 times, results are only limited by computing power, especially with the good performance of the Woodcrest. Figure 3 shows results with matrix sizes of 800x800 However, prefetching can be usefull in cases when we want to use a minimal amount of cache for the blocking, as the reuse count of a block directly relates to the block size.

We also try out the use of the prefetch thread for joining blocks by copying. We introduced different padding between matrix rows to provoke slower performance because of cache misses. With the prefetch thread, we always could improve the performance by around 3 %. We are investigating, if this advanced use of the prefetch thread can be used in other beneficial ways.

To summarize, the main benefit of block prefetching is that it equalizes changing bandwidth requirements of codes, thus avoiding memory bound program phases. This especially complements cache blocking techniques.

5 Future Work

An important real application are multigrid solvers, which typically are memory bound because of limited blocking possibilities on the finest grid level. In the DiME project, a well-known 3D multigrid solver was optimized with standard techniques [KRTW02], and recently further improved using much time and architecture knowledge on the Itanium [Stü06]. We hope to get similar performance without much changes of the original code, with the help of the prefetch thread.

Another area for research are tools to assist programmers to find code regions where insertion of block prefetch and copy requests is beneficial. To achieve this, in-depth code analysis with runtime-instrumentation will be used.

By using the second core for prefetching, we could show significant improvements in mem-

ory bound code. With more cores on one die, this technique, especially with special, small prefetch cores, can be quite useful in the future.

References

[Bey04] Kristof Beyls. *Software Methods to Improve Data Locality and Cache Behavior*. PhD thesis, June 2004.

[BS97] F. Bodin and A. Seznec. Skewed associativity improves program performance and enhances predictability. *IEEE Transactions on Computers*, May 1997.

[DDS95] F. Dahlgren, M. Dubois, and Per Stenström. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7), July 1995.

[KLea04] D. Kim, S. S. Liao, and P. H. Wang et al. Physical experimentation with prefetching helper threads on Intels hyper-threaded processors. In *Proc. of Intl. Symposium on Code Generation and Optimization*, 2004.

[KRTW02] M. Kowarschik, U. Rüde, N. Thürey, and C. Weiß. Performance Optimization of 3D Multigrid on Hierarchical Memory Architectures. In *Proc. of the 6th Int. Conf. on Applied Parallel Computing (PARA 2002)*, volume 2367 of *Lecture Notes in Computer Science*, pages 307–316, Espoo, Finland, June 2002. Springer.

[MKP05] O. Mutlu, H. Kim, and Y.N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd Int. Symposium on Comp. Architecture*, 2005.

[Stü06] M. Stürmer. Optimierung eines Mehrgitteralgorithmus auf der IA64 Rechnerarchitektur. Master's thesis, Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, 2006.

[WT06] J. Weidendorfer and C. Trinitis. Cache optimizations for iterative numerical codes aware of hardware prefetching. In *Applied Parallel Computing: 7th International Conference, PARA 2004, Lyngby, Denmark, June 20-23, 2004. Revised Selected Papers*, volume 3732, pages 921 – 927. Springer, 2006.