Off-loading Application controlled Data Prefetching in numerical Codes for Multi-Core Processors

J. Weidendorfer*
Institut für Informatik, Technische Universität München,
D-85747 Garching bei München, Germany
E-mail: Josef.Weidendorfer@cs.tum.edu
*Corresponding author

C. Trinitis
Institut für Informatik, Technische Universität München,
D-85747 Garching bei München, Germany
E-mail: Carsten.Trinitis@cs.tum.edu

Abstract: An important issue when designing numerical code in High Performance Computing is cache optimization in order to exploit the performance potential of a given target architecture. This includes techniques to improve memory access locality as well as prefetching. Inherent algorithm constrains often limit the first approach, which typically uses a blocking technique. While there exist automatic prefetching mechanisms in hardware and/or compilers, they can not complement blocking with additional prefetching.

We provide an infrastructure for off-loading application controlled prefetching on a chip multiprocessor, allowing to further improve numerical code already optimized by standard cache optimization. Clear benefits are shown for real workloads on existing hardware.

Keywords: chip multiprocessing, multi-core, cache optimization, prefetching.

Reference to this paper should be made as follows: Weidendorfer, J., Trinitis, C. (2006) 'Off-loading application controlled data prefetching in numerical codes for multi-core processors', Int. J. High Performance Computing and Networking, Vol. 1, Nos. 1/2/3, pp.64–74.

Biographical notes: J. Weidendorfer is a research assistant in the group of Prof. Dr. A. Bode at the Institut für Informatik, Technische Universität München, Germany. After a grant at BMW, working on load balancing for parallel FEM simulation, he received his PhD degree in 2003. Since then he does research in performance analysis and cache optimization with an increasing focus on multi-core architectures as member of the Munich Multi-core Initiative (MMI).[*]
C. Trinitis is a senior scientist and head of the research group on parallel and distributed computer architectures in the same group of Prof. Dr. A. Bode. He received his PhD in electrical engineering in 1998. His research interests comprise computer and microprocessor architectures with a special focus on multi-core systems and cluster computing.

1 INTRODUCTION

It is expected that within the next few years, multi-core processors — also known as *chip multiprocessing* (CMP) — will be common and pervasive in all computing systems. One reason is the exhausted instruction level parallelism (ILP) already available in current processors, making further enhanced superscalarity and out-of-order units almost useless despite increased complexity. The other reason are obvious limits for increased clock rates, which make for

[*]`http://mmi.in.tum.de`

enormous power consumption because of leakage currents. However, Moore's Law, revisited in Mollick (2006), seems to hold true for at least another decade, providing vast numbers of transistor capacities. To be able to exploit these resources for improved compute power, one solution is to increase thread level parallelism (TLP). The result are chip multiprocessors which are expected to have hundreds of cores within the next five years. However, with multi-core processors, the gap between main memory and CPU performance will grow at a higher pace, as the cores have to share one connection to the off-chip main memory.

Kowarschik and Weiß (2003) show that cache optimization is an important technique for numerous numerical codes to exploit the performance of current processors. A standard strategy tries to improve the locality of memory accesses, i.e. it improves the probability that when a data access occurs, the data already resides in the cache. However, sometimes algorithms inherently limit the locality that can be obtained 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 can even get worse. Therefore, a rule is to load the required data as shortly as possible before its actual use. This type of prefetching can often be done automatically in hardware, e.g. by detecting stream accesses in the cache controller, as shown by Dahlgren et al. (1995), or by speculatively pre-executing the code using run-ahead execution, a technique proposed in Mutlu et al. (2005). Kim et al. (2004) show that the latter also can be done in software using a so-called *helper thread* which is automatically generated by the compiler. In addition, some compilers allow the use of prefetch directives, inserted into the source code: This way, software prefetch instructions are compiled into the binary. These instructions — available on most processors — fetch a cache line containing a given address from main memory into the processor cache.

The abovementioned "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, prefetching the data for a subsequent program phase with high bandwidth requirements is beneficial. This approach needs support from the programmer to provide the best locations in the code for this kind of prefetching. However, the way how prefetching actually is accomplished is not specified:

- With current processors, software prefetch instructions can be used which have to be interleaved into the main application. For efficiency, this often adds further nested loops in numerical kernels, making the code complex. Often, assembler coding is required.
- Prefetching can be off-loaded from the main application to another entity on the chip. With the advent of multi-core processors, an elegant possibility is to use another core for executing prefetch request of the main application. However, this approach requires shared caches between the prefetching and the application thread. Another ideal solution would be to provide a small programmable unit in the cache controller of a core, allowing for prefetching of arbitrary access patterns.

The usage of a general purpose core for prefetching seems to be wasting resources. However, such a core often can not be used for useful work if another core already saturates the memory bus. For example, our third benchmark, the red/black Jacobi solver, is memory bounded; parallelization within the red and black sweeps can easily be accomplished with OpenMP, as there is no data dependency inside sweeps. However, preliminary runtime measurements have shown that the simple sequential version always was significantly faster than any execution with multiple threads. As such a comparison is outside of the scope of this paper, we will skip a more detailed analysis here. However, it can be expected that heterogeneous multi-core processors will be available in the future, where few small cores could be specialized for optimized memory transfer tasks.

This article provides insights into the benefits of application controlled prefetching which is off-loaded to another execution unit of multi-core processors. For this, we use the latest Intel processors which are equipped with level 2 caches shared between two cores. Although the processors are not designed for this technique, requiring tight synchronization and an unlimiting cache coherency implementation, we show quite promising results for real-world code. The next chapter will start with an overview of some related work. In chapter 3, the basic infrastructure is given for off-loading prefetching to a second core. Finally, we will present the applications and measurements results.

2 STATE OF THE ART

If a programmer intends to implement application controlled prefetching on processors which only allow a single thread to run at one time (neither multi-threaded nor multi-core), software prefetch instructions need to be interleaved with the regular application code. In Weidendorfer and Trinitis (2006), we have shown that in simple cases, this is possible and beneficial. To minimize instruction overhead, only one prefetch request should exist for each cache line, enforcing the coding of further loop levels. However, even with modest complex code, these additionally required deeper loop nests make this approach counterproductive. This effect occurs because the added instructions and branches touch the limit of the decoding stage. Yet, Beyls (2004) 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 will be shown, our off-loading technique also allows to re-layout in a quite efficient way.

With multi-threading, prefetching can be carried out in another simultaneous running thread: however, our experiments with *Hyperthreading* on Intel processors showed problems, because a running hyper-thread slows down the other thread to 70 percent of its original performance, making it difficult to obtain any benefits. However, Kim et al. (2004) have shown that is is possible to achieve speedups this way.

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 by Bodin and Seznec (1997), where it is compared to a hardware approach called *skewed associativity* to minimize conflict misses.

Prefetching is in most cases done by hardware, which tries to automatically detect memory access patterns like streaming loads, as described by Intel (2006) for their processors. A lot of algorithms have been proposed in the past; however, more complex detection schemes can require large table structures, and aggressive prefetching has the unwanted side effect of potentially polluting the cache with non-required data.

3 OFF-LOADING INFRASTRUCTURE

The best way to implement prefetching is to disturb the application code with a minimum impact. This can be obtained by defining prefetch requests which load multiple data blocks or linked structures within one command. These requests could be implemented as part of the instruction set (ISA) of the processor. To the best of our knowledge, processors with such instructions do not yet exist. However, there exist multi-core processors in which cores share a cache. For example, the Intel Core Duo (Yonah) processor and processors with Intel's new Core microarchitecture such as Core 2 Duo, Xeon 5100 and 5300 series, provide this feature. The former have two, the latter have four cores inside a socket. This cache organization allows for one core, running the main application, to trigger a large prefetch request by issuing a single command to its sibling core, which actually executes the prefetching into the shared cache. This ensures the abovementioned minimal impact.

Fig. 1 shows the basic structure of the proposed prefetch infrastructure. It is actually 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

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

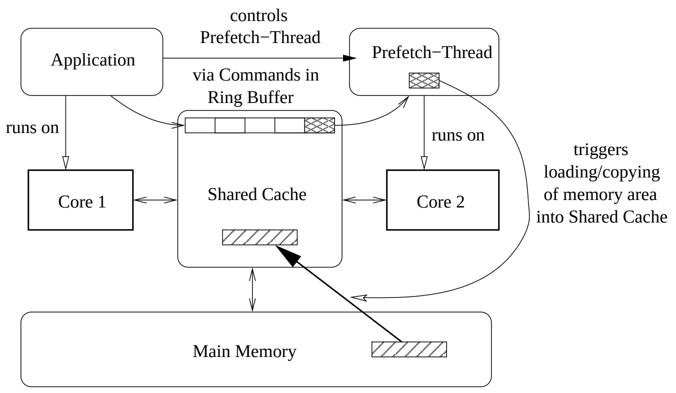

Figure 1: Infrastructure for off-loading prefetching to another processor core

application can still be beneficial. 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. 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 next to the core which runs the main application, such that both cores share a common cache. In a multiprocessor machine on Linux, this can be achieved by setting a thread's CPU affinity mask to bind it to a given processor/core, thus forcing the required 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 sufficient 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 hampering 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 case of the Intel CPUs, we did not notice any benefit in throttling the prefetch thread. A more detailed analysis is subject to ongoing research.

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 a prefetch operation; however, it ensures that 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 additional traffic on the memory bus. The copy operations allow data from memory to be rearranged in the

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]
```

Figure 3: Synthetic one-dimensional blocking benchmark

cache for more compact, and therefore faster, access. Additionally, 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 should benefit from sequential access, like the hardware stride prefetcher implemented for the L1 cache controller in Intel's Core microarchitecture, as described in Intel (2006).

Prefetching by copying pollutes twice the space of the data when the copy is done. Therefore, the later access benefits have to compensate this effect in order 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 efficient as a pure block prefetch. The Intel 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 BENCHMARKS

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 the cache. In Weidendorfer and Trinitis (2006), we already used similar code. Fig. 2 shows the scenario with a blocking factor of two. Here, we have interleaved 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. Fig. 3 shows the benchmark code with the request to the prefetch thread.

The second application we did measurements with is matrix multiplication: With appropriate two-dimensional blocking, e.g. reusing a matrix block 50 times, results are only limited by compute power. As one can also see from the computational complexity of $O(N^3)$, in contrast to memory transfer complexity of $O(N^2)$, the memory bandwidth typically is not a limiting issue for matrix multiplication. However, prefetching can be beneficial for 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.

The last application we did measurements for comes from the important class of iterative solvers for differential equations: Here, programmers typically have to deal with a two- or three-dimensional grid in the model space, which is represented by a corresponding matrix in memory, providing physical quantities at every grid point for the given equation to solve. Sophisticated multi-grid solvers highly reduce the number of full sweeps over the whole grid by going forth and back between different resolutions of the grid. On the finest level, it is typically sufficient to carry out two or four smoothing sweeps for a larger multi-grid cycle. However, memory bandwidth requirements remain high, demanding for cache optimization strategies. Weiß et al. (2002) did a lot of detailed optimization strategies, with even using assembly code, within the scope of the DiME project.

In this work, we show that prefetching can improve the performance of code without making it significantly more complex. We have implemented a simple 2-dimensional red/black Jacobi solver, and started with implementing a few optimizations without any prefetching. First, memory access behavior can be improved by separating the red and black cells into two separate matrices. Instead of a matrix with S intermixed red and black cells, which needs to be fully written back to memory on every sweep, we now have two regions of size $S/2$ with pure read or write access, respectively. Net effect is 1/4 less memory transactions[2]. Furthermore, we can use so-called *non-temporal writes* which bypass the cache, saving another 1/4 of the original memory transactions. The latter unfortunately complicates the code because a non-temporal write requires the explicit use of an x86 instruction from the SSE-2 subset with a 16-byte wide data type. In another opti-

[2]Packing of equally colored cells additionally enables the use of SIMD instructions. However, we do not take advantage of this, as the Intel compiler was not able to do this automatically.

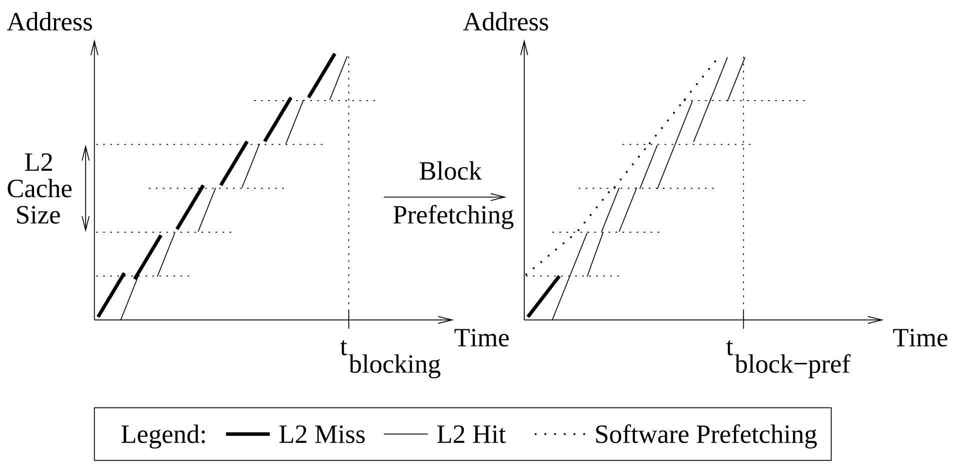

Figure 2: 1D-blocking enhanced with simultaneous prefetching of next memory block.

mization, a red and a black sweep can be interleaved into one pass over the original matrix. This — similar two the previous optimizations — results in 1/2 of the memory transactions, without separating the red and black cells. Finally, we added prefetching for the original R/B solver as well as for the interleaved version. As can be seen in the next section, we are able to show impressive improvements.

5 RESULTS

Measurements were carried out on

- a 2.16 GHz Intel Core Duo (Yonah) processor, two cores sharing a 2 MB level 2 cache. Compiler flags `-O3 -xP`.
- a pre-series Xeon 5100 system with two 2.6 GHz Intel Core processors (Woodcrest), each with two cores sharing a 4 MB level 2 cache. Compiler flags `-O3 -xT`.
- a Xeon 5315 system with two 2.6 GHz Intel Quad Core processors (Clovertown), each with four cores where two of them share a 4 MB level 2 cache each. Compiler flags `-O3 -xT`.

The first system is 32 bit, the latter ones are 64 bit.

Results of the synthetic benchmark can be seen in Fig. 4 (for Yonah), Fig. 5 (for Woodcrest), and Fig. 6 (for Clovertown). The figures show the reachable transfer rate to memory as seen from a processor core, against the number of interactions done over one memory block. Results are shown without prefetching, with interleaved prefetching (quite complex code), and with prefetching off-loaded to the other core. While the interleaved prefetching seems to perform best, this technique is not adequate for real application code. In all cases, usage of a prefetching thread comes quite close. For the Yonah processor, the prefetching thread is even the best solution. The reason is probably because the limited number of available registers in 32bit x86 assembler leads to worse code for the interleaved version.

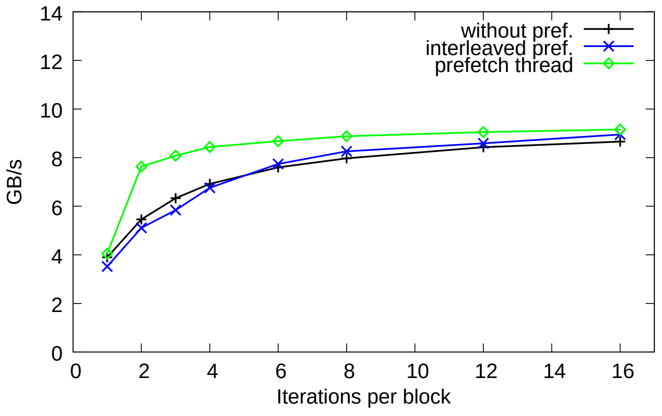

Figure 4: 1D-blocking results on Yonah.

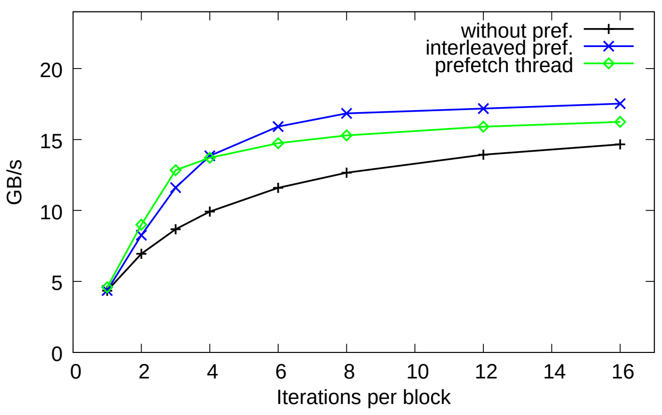

Figure 5: 1D-blocking results on Woodcrest.

The results of the matrix multiplication code are shown in Fig. 7, Fig. 8, and Fig. 9. The reachable compute power is shown against matrix size ranging from 20^2 to 1000^2. The largest configuration has a memory requirement of 24 MB total for the matrices.

In each case, the naive version and blocked variants are given, partly with prefetching enabled. To present ben-

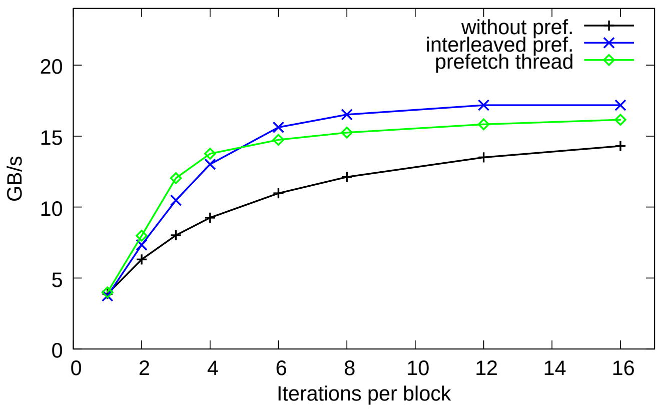

Figure 6: 1D-blocking results on Clovertown.

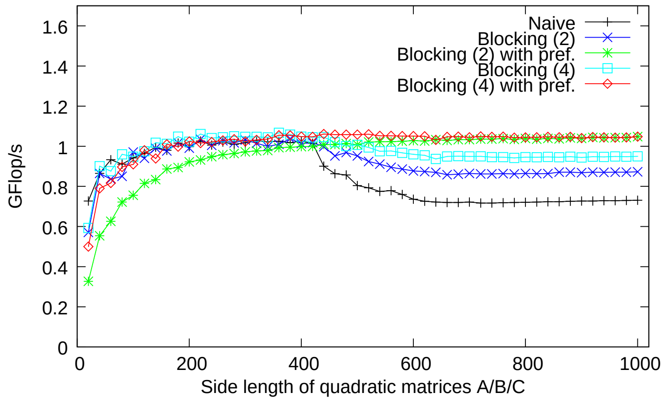

Figure 7: Matrix multiplication results on Yonah.

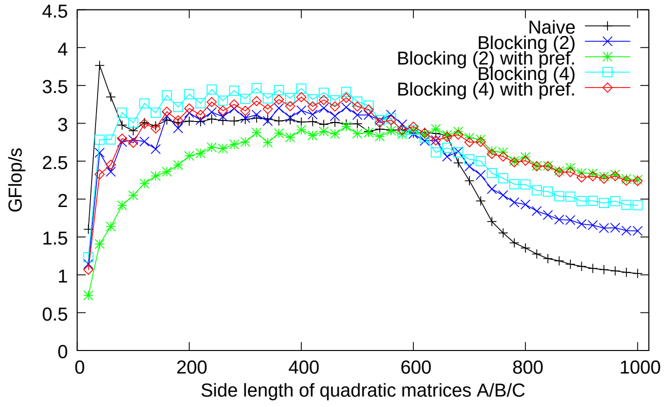

Figure 8: Matrix multiplication results on Woodcrest.

efits of prefetching, small block sizes (2 and 4 rows) are provided, leading to 16 KB and 32 KB used cache space only. This shows that prefetching shines even with very limited cache available.

Finally, performance results for the red/black Jacobi solver are shown in Fig. 10, Fig. 11, and Fig. 12, respectively. Both the versions using non-temporal writes ("non-temp. packed R/B") as well as the interleaving optimization ("R/B interleaved") provide better perfor-

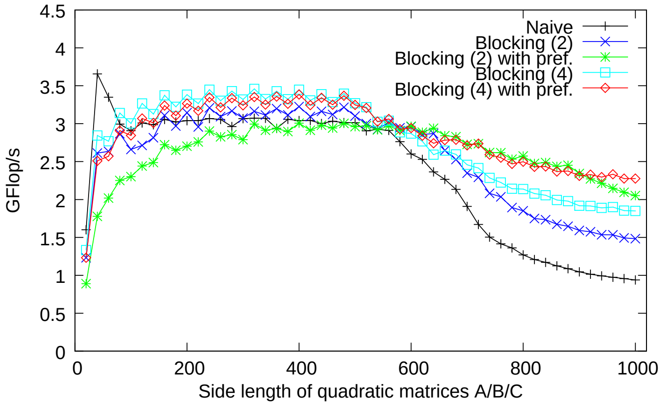

Figure 9: Matrix multiplication results on Clovertown.

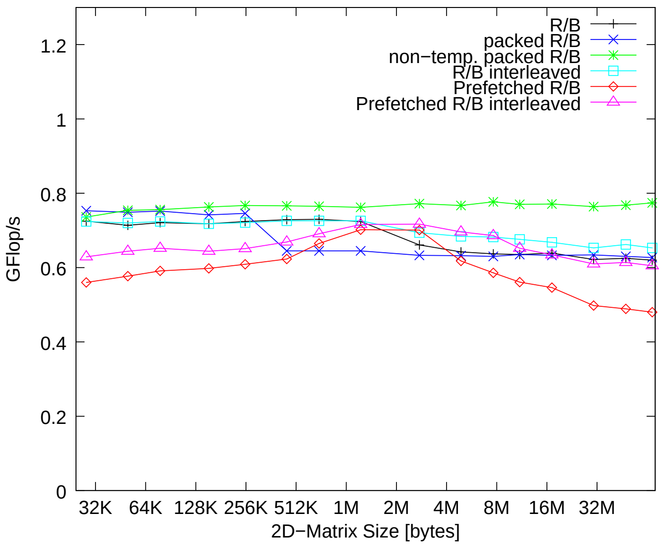

Figure 10: Red/Black Jacobi results on Yonah.

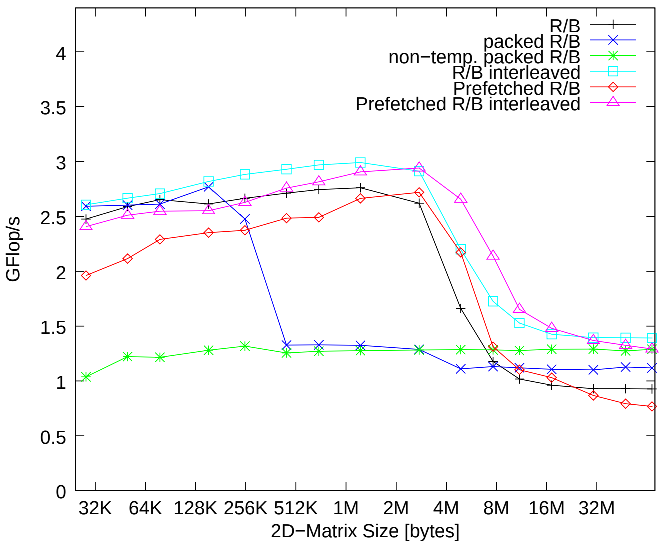

Figure 11: Red/Black Jacobi results on Woodcrest.

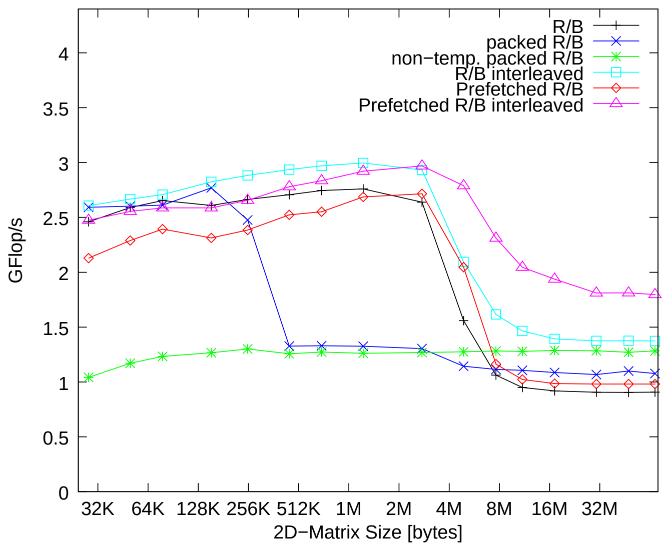

Figure 12: Red/Black Jacobi results on Clovertown.

mance than the original R/B solver for all processors. As stated in the last chapter, they should only need half as many memory transactions than the original version. However, performance is far from doubled despite the memory-boundedness of the benchmark.

An interesting result on the Yonah processor is that the version bypassing the cache on write is the fastest over all sizes, even if the data structure fully fits into the cache. Obviously, the compute power of this processor is quite weak in comparison with the available memory transfer rate (at least for this application). On the other side, prefetching on this processor is not very helpful. This is probably because of a limited ability for sharing the data in an efficient manner among the cores, despite of its shared L2 cache.

On the other hand, Intel's Core microarchitecture has more than enough compute power to turn the memory connection into a bottleneck: When data fits into the cache, results are way higher than with large matrices. As stated before, when interleaving red and black sweeps, one would expect double performance on this processor. However, without prefetching, this expectation is not fulfilled. Obviously, memory accesses done by the main application are done in a less efficient way. In contrast, the prefetching thread can exploit the memory bandwidth much better. Comparing the Woodcrest and Clovertown processors, they show quite similar results. However, prefetching degrades on very large matrix sizes on the Woodcrest. The Clovertown, while seemingly consisting of two Woodcrest dies on one chip, obviously has an improved cache architecture: prefetching on the interleaved version gives a constant improvement of around 30% for very large matrixes, thus reaching the theoretically possible doubled performance in comparison to the original R/B version.

Finally, a strange effect can be observed for the Core microarchitecture and the "packed R/B" versions: performance strongly degrades already at a size of 512 KB although this easily should fit into the L2 cache of 4 MB. It is assumed that memory transactions can contribute to this effect. The relation of this degradation to the used optimization technique (separating read and written regions) is not clear; a possible conclusion would be that the Core architecture holds a maximum of 256 KB of purely written data in the L2 cache[3].

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 and sweep interleaving techniques.

6 CONCLUSIONS

An important class of applications in High Performance Computing are multi-grid solvers, which are typically memory bound because of limited blocking possibilities on the finest grid level. Within the DiME project, a well-known 3D multi-grid solver was optimized with standard techniques by Kowarschik et al. (2002), and recently further improved with a lot of effort on the Itanium processor by Stürmer (2006). It is clear that the same performance as handcrafted detail-optimized assembly code can hardly be obtained by our approach. However, taking its code clearness and short development time into consideration, the technique using a prefetch thread is quite competitive, as has been shown in this article. In this work, we dedicated one general purpose core to prefetching. Although such a core could be used for parallelization of the main application, preliminary results show that such an approach is even worse than the sequential code without prefetching, when the application is memory bounded.

A further area of research are tools that assist programmers to find code regions where insertion of block prefetch and copy requests can be 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 memory bounded code. With more cores on one die, this technique, especially with special, small prefetch cores, can be quite useful in the future.

ACKNOWLEDGMENT

We would like to thank Intel Corporation for providing us pre-series versions of their latest multi-core processor systems for this work.

[3]Interestingly, this is one way of the 16-way associativity used with this cache.

REFERENCES

Beyls, K. (2004). *Software Methods to Improve Data Locality and Cache Behavior*. PhD thesis.

Bodin, F. and Seznec, A. (1997). Skewed associativity improves program performance and enhances predictability. *IEEE Transactions on Computers*.

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

Intel (2006). *Intel 64 and IA-32 Architectures: Software Developer's Manual*. Denver, CO, USA.

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

Kowarschik, M., Rüde, U., Thürey, N., and Weiß, C. (2002). 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. Springer.

Kowarschik, M. and Weiß, C. (2003). An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In Meyer, U., Sanders, P., and Sibeyn, J., editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*, pages 213–232. Springer.

Mollick, E. (2006). Establishing moore's law. *IEEE Annals of the History of Computing*, 28(3):62–75.

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

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

Weidendorfer, J. and Trinitis, C. (2006). 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.

Weiß C., Hellwagner, H., Stals, L., and Rüde, U. (2002). Data Locality Optimizations to Improve The Efficiency of Multigrid Methods. Technical Report 02–1, Lehrstuhl für Informatik 10 (Systemsimulation), University of Erlangen-Nuremberg, Germany.