Cache Optimizations for Iterative Numerical Codes Aware of Hardware Prefetching

Josef Weidendorfer, Carsten Trinitis

Technische Universität München, Germany
{weidendo,trinitic}@cs.tum.edu

Abstract. Cache optimizations typically include code transformations to increase the locality of memory accesses. An orthogonal approach is to enable for latency hiding by introducing prefetching techniques. With software prefetching, cache load instructions have to be inserted into the program code. To overcome this complexity for the programmer, modern processers are equipped with hardware prefetching units which predict future memory accesses in order to automatically load data into cache before its use.
For optimal performance, it seems advantageous to combine both prefetching approaches. In this contribution, we first use a cache simulation enhanced with a simple hardware prefetcher to run code for a 3D multigrid solver. Cache misses which are not predicted by the prefetcher can be located in simulation results, and selectively, software prefetch instructions can be inserted. However, when performance of a code section is limited by available bandwidth to main memory, this simple strategy will fail. Thus, we use *Block Prefetching*, an extension of the standard blocking strategy. Meassurements show its potential.

1 Introduction

Cache optimizations typically include code transformations to increase the locality of memory accesses: standard strategies are loop blocking and relayouting of data structures, e.g. splitting or joining of arrays and padding [12]. An orthogonal approach is to enable for latency hiding by introducing prefetching techniques; i.e., by ensuring that any data is loaded early enough before it is actually used. Software prefetching enables this by inserting cache load instructions into the program code. However, the use of such instructions consumes both decoding bandwidth and hardware resources for the handling of outstanding loads. Because of this and the added complexity of manually inserting prefetch instructions, modern processers like Intel Pentium 4, Pentium-M [10] or AMD Athlon, are equipped with hardware prefetch units which predict future memory accesses in order to load data into cache in advance.

Both prefetch approaches can be combined. Usually, there will be parts of code where hardware prefetching is doing a fine job, and parts where manual insertion of prefetch instructions is needed. The two cases can be distinguished by doing measurements with hardware performance counters. To this end, we

use the Intel Pentium-M processor hardware, which has performance counters meassuring the effect of its hardware prefetch unit [10]. This approach has some drawbacks: the actual prefetch algorithm implemented inside the processor is unknown, and thus, manual insertions of prefetch instructions will be specific to one processor. Therefore, additional measurements are done by simulating a known hardware prefetch algorithm. By choosing a simple stream detection, we are sure that program code where this prefetch algorithm is working fine, is also running fine with any existing hardware prefetcher. Additionally, we can compare results with and without hardware prefetching.

There is however a case where prefetching alone can not help: When performance of a code section is limited by available bandwidth to main memory, prefetching can not do any better. The first step to improve this situation is to use loop blocking [12]: instead of going multiple times over a large block of data not fitting into the cache, we split up this block into smaller ones fitting into the cache, and go multiple times over each small block before handling the next one. Thus, only in the first run over a small block has to fetch data from main memory. Still, this first run exhibits the bandwidth limitation. But as further runs on the small block do not use main memory bandwidth at all, this time can be used to prefetch the next block. Of course, the block size needs to be corrected in a way that two blocks fit into cache. We call this technique *Block Prefetching*.

For this study, a 3D multigrid solver is used as application. The code is well known, and various standard cache optimizations were applied [11]. The main motivation for this work is the fact that standard cache optimizations do not work as well as expected on newer processors [17]. On the one hand, hardware prefetchers seem to help in the non-optimized case, on the other hand, they sometimes seem to work against manual cache optimizations: e.g. blocking with small loop intervals, especially in 3D, leads to a lot of unneeded activity from a hardware prefetcher with stream detection.

In the next chapter, some related work is presented. Afterwards, we give an survey of the cache simulator used and the hardware prefetch extension we added. We show measurements for our multigrid solver, using statistical sampling with hardware performance counters, and the simulator without and with hardware prefetching. Finally, for a simple example we introduce block prefetching and give some real measurements. Since adding block prefetching to the multigrid solver is currently work in progress, results will be given in a subsequent publication.

2 Related Work

Regarding hardware prefetchers, [3] gives a good overview of well known algorithms. Advanced algorithms are presented in [1]. Still, in hardware manuals of processors like Pentium-4 [10], there is only a vague description of the prefetch algorithm used.

Simulation is often used to get more details on the runtime behavior of applications or for research and development of processors [16],[9]. For memory access behavior and cache optimization, it is usually enough to simulate the cache hierarchy only like in MemSpy [14] or SIGMA [7]. The advantage of simulation is the possiblity to get more useful metrics than plain hit/miss counts like reuse distances [4] or temporal/spatial locality [2].

For real measurements, we choose statistical sampling under Linux with OProfile [13]. With instrumentation, we would get exact values, e.g. using Adaptor [5], a Fortran source instrumentation tool, or DynaProf [8]. The latter uses DynInst [6] to instrument the running executable with low overhead. All these tools allow to use the hardware performance counters available on a processor.

3 Simulation of a simple Hardware Prefetcher

Our cache simulator is derived from the cache simulator Cachegrind, part of the Valgrind runtime instrumentation framework [15], and thus, it can run unmodified executables. We describe the simulator with its on-the-fly call-graph generation together with our visualization tool KCachegrind in more detail in [18]. With a conversion script, the latter is also able to visualize the sampling data from OProfile.

The implemented hardware prefetch algorithm gets active on L1 misses: for every 4KB memory page, it checks for sequential loads of cache lines, either upwards or downwards. When three consecutive lines are loaded, it assumes a streaming behavior and prefetches the cache line which is 5 lines in advance. As the simulator has no way to do timings, we simply assume this line to be loaded immediately. Although this design is arguable, it fulfills our requirement that every sensible hardware prefetcher should at least detect the same streaming as our prefetcher.

The following results have been meassured for a cache-optimized Fortran implementation of multigrid V(2,2) cycles, involving variable 7-point stencils on a regular 3D grid with 129^3 nodes. Compared to the standard version, the simulated as well as real meassurement show little more than half the number of L2 misses / L2 lines read in. As only 2 smoothing iterations can be blocked with the given multigrid cycle, this shows that the blocking works quite well. In search for further improvement using prefetching, table 1 shows the number of simulated L2 misses, first column with hardware prefetching switched off, second column with hardware prefetching switched on (i.e. only misses that are not catched by the prefetcher), and third column the number of prefetch actions issued. For real measurements, the Pentium-M can count L2 lines read in because of L2 misses alone, as shown in column 4, lines read in because of a request from the hardware prefetcher only in column 5. The last column shows the number of prefetch requests issued by the hardware. All numbers are given in millons of events (mega events). The first line gives a summary for the whole run, the next 2 lines splitted up by top functions: RB4W is the red-black gauss-seidel smoother, doing the main work, and RESTR is the restriction step in the multigrid cycle.

Table 1. Measurements for the multigrid code [MEv].

	Simulated			Real		
	L2 Misses		Pref.	L2 Lines In		Pref.
	Pf. Off	Pf. On	Requests	due to Misses	due to Pref.	Requests
Summary	361	277	110	241	130	373
RB4W	233	226	-	201	47	270
RESTR	108	37	-	28	76	92

Simulation results show that the function RESTR, is hardware-prefetcher friendly, as L2 misses go down by a large amount when the prefetcher is switched on in the simulation. In contrast, for the first function, RB4W, our simulated prefetcher does not seem to work. Looking at actual hardware, the prefetcher in the Pentium-M works better than our one: it reduces the numbers of L2 misses (col. 4) even more than our one (col.2), especially it is doing a better job in RB4W (but not to good, either). This shows that our simulation can show the same trend as will be seen in real hardware regarding friendlyness to stream prefetching.

4 Block Prefetching

The idea of Block Prefetching is based on loop blocking. When a program has to work multiple times on a huge array not fitting into cache, data has to be loaded every time from main memory. When data dependencies of the algorithm allow to reorder work on array elements such that the multiple passes can be done consecutively on smaller parts of the array, Blocking is possible. This improves temporal locality of memory accesses, i.e. accesses to the same data happen near to each other. By using block sizes that fit into cache (usually L2), this means that only for the first run over a small block, data has to be fetched from main memory. Fig. 1 depicts this scenario on a 1-dimensional array. The time $t_{blocking}$ is substantially less than t_{orig}: with blocking, the second iteration of each block can be done with L2 bandwidth $L2B$ instead of memory bandwidth MB, and thus gets a speedup of $\frac{MB}{L2B}$, i.e.

$$t_{blocking} = t_{orig} * (\frac{1}{2} + \frac{L2B}{2 * MB}).$$

As further runs on the small block do not use main memory bandwidth at all, this time can be used to prefetch the next block. This needs block size corrected in a way that two blocks fit into cache. For an inner block, we assume it to be already in the cache when work on it begins. When there are N runs over this block, we can spread prefetching of the next block equally over these N iterations. For the multigrid code, $N = 2$, i.e. the pressure on memory bandwidth is almost cut in half. Fig. 2 shows this scenario. If we suppose that the needed bandwidth for prefetching is not near the memory bandwidth limit, almost all the time we can work with the L2 bandwidth $L2B$, depending on the relation of the first

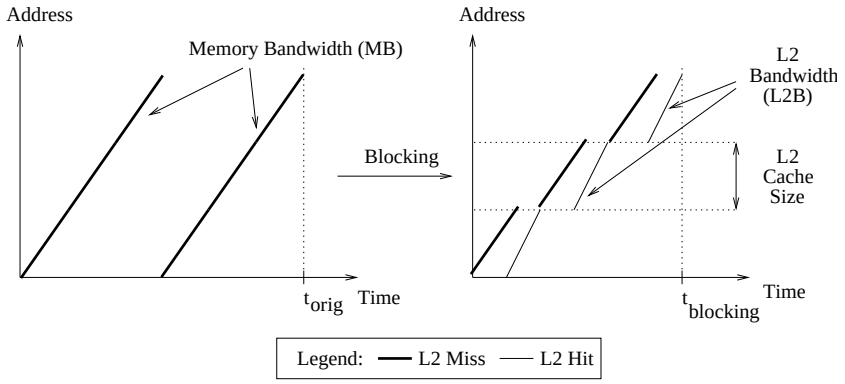

Fig. 1. The blocking transformation.

block size to the whole array size as shown in the figure. Note that for the first block, memory pressure will be even higher than before because in addition to the first block, half of the second block is prefetched simultaneously.

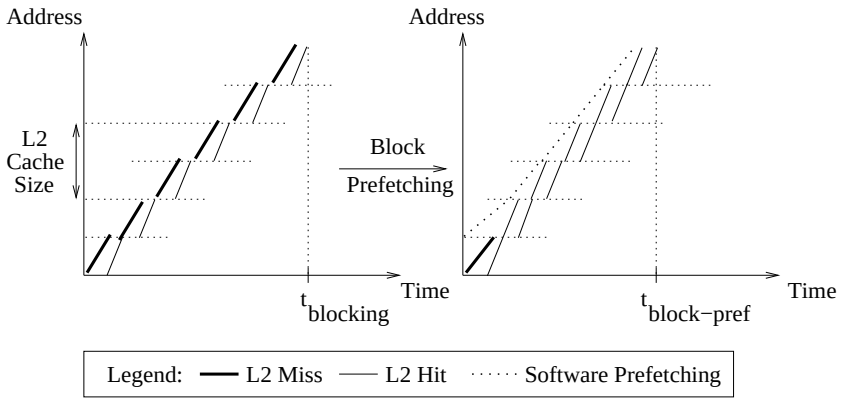

Fig. 2. The block prefetching transformation.

Table 2 gives results for a microbenchmark with 1-dimensional blocking for different numbers of N. With pure blocking, $N = 1$ make no sense, as data is not reused. Still, block prefetching, which is the same as normal prefetching in this case, obviously helps. Again, these results were meassured on a 1.4 GHz Pentium-M with DDR-266 memory, i.e. a maximum memory bandwith of 2.1 MB/s. The benchmark was compiled to use x87 instructions only (no MMX or SSE).

We show results with two modifications of the benchmark: the first does one floating point operation per load of a double, the second does two operations.

Table 2. Results of a microbenchmark.

	Without Block Pf.				With Block Pf.			
N	1	2	3	10	1	2	3	10
1 Flop/Load								
Runtime [s]	5.52	3.92	3.45	2.73	4.89	2.67	2.50	2.44
MFlop/s	194	274	311	393	220	402	430	440
Netto [GB/s]	1.48	1.04	0.79	0.30	1.68	1.53	1.09	0.34
Lines In b/o Misses [MEv.]	4.2	2.1	1.4	0.4	134	67	45	13
Lines In b/o Pref. [MEv.]	130	65	44	13	0.45	0.54	0.27	0.15
2 Flops/Load								
Runtime [s]	7.20	5.24	4.81	3.23	6.86	4.29	3.60	2.84
MFlop/s	298	410	446	665	313	501	597	756
Netto [GB/s]	1.14	0.78	0.57	0.25	1.19	0.95	0.76	0.29

For each benchmark and N, we show the runtime in seconds, and the achieved (netto) bandwidth from main memory. The netto bandwidth is calculated from the runtime and the known amount of data loaded. A brutto bandwidth that gives the bandwidth needed from memory to L2, can be calculated from memory read burst transactions, meassured by a performance counter: the benchmark only loads data, and cache line loads are translated to burst requests. The brutto bandwidth is always between 50 and 100 MB/s higher than the netto bandwidth in this benchmark, and therefore not shown in the table.

Runtime figures show that speedups of up to 46 percent are possible ($N = 2$, 1 Flop/Load). To see the effect on the number of L2 lines read in on the one hand because of misses, and on the other hand because of hardware prefetch requests, we show these numbers for the first benchmark. Obviously, block prefetching virtually switches off the hardware prefetching.

5 Conclusion and Future Work

In this paper, we have shown the usefulness of introducing a simple hardware prefetch algorithm into a cache simulator. By comparing a simulation with hardware prefetching switched off with the simulation of the same program run, but prefetching switched on, one can see the positions in the code where insertion of software prefetching instructions is useful. We extend this result by presenting the block prefetching technique which is able to lower the pressure on memory bandwidth.

Still, for simulation with hardware prefetching to be really worthful, visualization possiblities have to be enhanced: even source annotation can not differentiate between iterations of the same loop body, as only sums are given. But it appears useful to be able to distinguish at least the first few iterations of a loop, as it is expected that prefetch characteristics change here. Thus, more detailed context separation of profile data is needed. The block prefetching technique has to be integrated into real world applications like our multigrid solver.

References

1. M. Bekerman, S. Jourdan, R. Romen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated Load-Address Predictors. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 54–63, May 1999.
2. E. Berg and E. Hagersten. SIP: Performance Tuning through Source Code Interdependence. In *Proceedings of the 8th International Euro-Par Conference (Euro-Par 2002)*, pages 177–186, Paderborn, Germany, August 2002.
3. Stefan G. Berg. Cache prefetching. Technical Report UW-CSE 02-02-04, University of Washington, Februar 2002.
4. K. Beyls and E.H. D'Hollander. Platform-Independent Cache Optimization by Pinpointing Low-Locality Reuse. In *Proceedings of International Conference on Computational Science*, volume 3, pages 463–470, June 2004.
5. T. Brandes. Adaptor. homepage. http://www.scai.fraunhofer.de/291.0.html.
6. B. Buck and J.K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14:317–329, 2000.
7. L. DeRose, K. Ekanadham, J. K. Hollingsworth, and S. Sbaraglia. SIGMA: A Simulator Infrastructure to Guide Memory Analysis. In *Proceedings of SC 2002*, Baltimore, MD, November 2002.
8. Dynaprof Homepage. http://www.cs.utk.edu/ mucci/dynaprof.
9. H. C. Hsiao and C. T. King. MICA: A Memory and Interconnect Simulation Environment for Cache-based Architectures. In *Proceedings of the 33rd IEEE Annual Simulation Symposium (SS 2000)*, pages 317–325, April 2000.
10. Intel Corporation. IA-32 Intel Architecture: Software Developers Manual.
11. 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.
12. M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*, pages 213–232. Springer, March 2003.
13. J. Levon. OProfile, a system-wide profiler for Linux systems. Homepage: http://oprofile.sourceforge.net.
14. M. Martonosi, A. Gupta, and T. E. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *Measurement and Modeling of Computer Systems*, pages 1–12, 1992.
15. N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, Boulder, Colorado, USA, July 2003. Available at http://developer.kde.org/˜sewardj.
16. V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1996.
17. N. Thürey. Cache Optimizations for Multigrid in 3D. Lehrstuhl für Informatik 10 (Systemsimulation), Institut für Informatik, University of Erlangen-Nuremberg, Germany, June 2002. Studienarbeit.
18. J. Weidendorfer, M. Kowarschik, and C. Trinitis. A Tool Suite for Simulation Based Analysis of Memory Access Behavior. In *Proceedings of International Conference on Computational Science*, volume 3, pages 455–462, June 2004.