Cache Memory Behavior of Advanced PDE Solvers

Dan Wallin, Henrik Johansson and Sverker Holmgren
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{dan.wallin, henrik.johansson, sverker.holmgren}@it.uu.se

Abstract

Three different partial differential equation (PDE) solver kernels are analyzed in respect to cache memory performance on a simulated shared memory computer. The kernels implement state-of-the-art solution algorithms for complex application problems, and the simulations are performed for data sets of realistic size.

The performance of the studied applications benefits from much longer cache lines than normally found in commercially available computer systems. The reason for this is that numerical algorithms are carefully coded and have regular memory access patterns. These programs take advantage of spatial locality and the amount of false sharing is limited. A simple sequential hardware prefetch strategy, providing cache behavior similar to a large cache line, could potentially yield large performance gains for these applications. Unfortunately, such prefetchers often lead to additional address snoops in multiprocessor caches. However, applying a bundle technique, which lumps several read address transactions together, this large increase in address snoops can be avoided. For all studied algorithms, both the address snoops and cache misses are largely reduced in the bundled prefetch protocol.

1 Introduction

The solution of partial differential equations (PDEs) is a central part of many important and demanding computations in science and technology. Despite increases in computational power and advances in solution algorithms, the computations are still often very time and memory consuming, and large shared memory servers have to be employed. This is especially true for PDEs arising from accurate models of complex, realistic problems. Hence, it is important that such problems can be solved efficiently on current and future parallel computers.

Designing a parallel computer system is an optimization problem where several parameters have to be taken into account, e.g. the cache miss rate, the number of address transactions sent on the network and the amount of data traffic generated. Improving one property often leads to worse behavior for another property. A longer cache line size normally efficiently reduces the number of cache misses in uniprocessors. In multiprocessors a longer cache line may lead to increased traffic as well as more cache misses if a large amount of false sharing occurs [25]. The preferred cache line size is therefore usually smaller in multiprocessors than in uniprocessors. The preferred cache line size is also very application dependent. Several papers have investigated the effects of cache line size on miss rate and traffic in multiprocessors [8, 11, 13, 29].

Most available multiprocessors are optimized for running commercial software, e.g. databases and server-applications, since the largest market share is within this field. In these applications, the data access pattern is very unstructured and a large amount of false sharing occurs [16, 28]. Popular benchmarks used when designing multiprocessors are the TPC-benchmarks [2] and the SPEC/OSG-benchmarks [1]. The computer vendors usually build multiprocessors with cache line sizes ranging between 32 and 128 B. This cache line size range gives rather good performance trade-offs between cache misses and traffic for commercial applications, but might not be ideal for scientific applications.

In this paper, we evaluate the behavior of three PDE solvers. The kernels are based on modern, efficient algorithms, and the settings and working sets are chosen to represent realistic application problems. The codes are written in Fortran 90, and parallelized using OpenMP directives. The study is based on full-system simulations of a shared memory multiprocessor. These problems are much more demanding than commonly used scientific benchmarks, e.g. found in SPLASH-2 [29], which usually implement simplified solution algorithms. The baseline computer model is set up to correspond to a commercially available system, the SunFire 6800 server. However, using a simulator, the cache coherence protocol can easily be modified to resemble alternative design choices for possible future designs. The simulations can also be useful for providing knowledge of possible performance bottlenecks to the programmer.

Based on the simulations, we conclude that the spatial locality is much better in these PDE kernels than in commercial benchmarks. Therefore, the optimal cache line size for these algorithms is larger than in most available multiprocessors. Spatial locality could be better explored also in a small cache line size multiprocessor using prefetching. A very simple sequential prefetch protocol, prefetching several consecutive addresses on each cache miss, give a cache miss rate similar to a large cache line protocol. However, the coherence and data traffic on the interconnect increase heavily compared to a non-prefetching protocol. We show that by using the bundling technique, previously published in [28], the coherence traffic can be kept under control.

2 The PDE solvers

The kernels studied below represent three important types of PDE solvers, used for compressible flow computations in computational fluid dynamics (CFD), radar cross-section computations in computational electromagnetics (CEM) and chemical reaction rate computations in quantum dynamics (QD). The properties of the kernels differ a lot with respect to the amount of computations performed per memory access, memory access patterns, amount of communication, and communication patterns.

2.1 The CFD kernel

The CFD kernel implements the advanced algorithm described by Jameson and Caughey for solving the compressible Euler equations on a structured grid [15]. The implementation is described in detail by Nordén et al [21].

The Euler equations are non-linear and the solution is vector valued. The computations are carried out on a structured, three-dimensional curvilinear grid where a finite volume discretization is used in combination with a flux-splitting scheme. The resulting system of equations is solved using a red-black ordered Gauss-Seidel-Newton (GSN) technique where a multigrid method is used to accelerate the convergence of the iterations.

The data used in the computations are highly structured. Each smoothing operation in the multigrid scheme consists of a GS iteration, which sweeps through an array representing the three-dimensional grid. Here, the values of the solution vector at six neighbor cells are used to update the values in each cell. To parallelize the GS iteration, a red-black ordering of the cells is introduced. Also, the updates are made in a symmetric way sweeping through the grid in a red-black-black-red order in each GS iteration. After smoothing, the solution is restricted to a coarser grid, where the smoother is applied again and the process is repeated recursively. At the coarsest grid, several smoothing operations are performed, and the solution is then gradually prolongated back to the finest grid. After each prolongation step, an additional smoothing sweep is performed.

The total work of the algorithm is heavily dominated by the computations performed within each cell for determining the updates in the smoothing operations. These local computations are quite involved, but the parallelization of the smoothing step is trivial. Each of the threads computes the updates for a slice of each grid in the multi-grid scheme. Communication is only performed in the smoothing operation and in the prolongation. The amount of communication is small, and the threads only communicate pair-wise.

2.2 The CEM Kernel

This kernel is part of an industrial Computational Electromagnetics (CEM) solver for determining the radar cross section of an object [9]. The solver utilizes an unstructured grid in three dimensions in combination with a finite element discretization. The resulting large system of equations is solved with a version of the conjugate gradient method [5]. The coefficient matrix is very sparse and unstructured. In each conjugate gradient iteration, a matrix-vector multiplication is performed with this matrix. This operation dominates the total computational work. The different stages in the iteration are parallelized in different ways. Local vector updates are trivially parallelized, and no communication is required. The inner product requires a global reduction, involving synchronization of all threads. Finally, the parallelization of the matrix-vector multiplication is performed such that each thread computes a block of entries in the result vector. The positions of the non-zero entries in the matrix determine which elements in the data vector that are used for computing a given entry in the result vector. Since the matrix is unstructured, the effect is that the data vector is accessed in a seemingly random way. However, the memory access pattern does not change between the iterations.

2.3 The QD Kernel

This PDE kernel is derived from an application where the dynamics of chemical reactions is studied using a quantum mechanical model with three degrees of freedom [22]. The solver utilizes a pseudo-spectral discretization in the two first dimensions, and a standard finite difference scheme in the third direction. In time, an explicit ODE-solver is used. The computational grid is structured and uniform. For computing the derivatives in the pseudo-spectral discretization, a standard convolution technique involving two-dimensional fast Fourier transforms (FFTs), local multiplications, and inverse FFTs is applied. The parallelization is performed such that the FFTs in the first dimension are performed in parallel and locally [27]. For the FFTs in the second dimension, a parallel transpose operation is applied to the solution data, and the local FFTs are applied in parallel again. The finite difference computations are fully parallel and local.

The communication within the kernel is concentrated to the transpose operation. This operation involves heavy all-to-all communication between the threads. However, the communication pattern is constant between the iterations in the time loop.

3 Simulation Environment

All experiments are carried out using execution-driven simulation in the full-system simulator Simics [18]. The modeled system uses the SPARC v9 ISA and implements a snoop-based invalidation MOSI (Modified, Owner, Shared, Invalid) cache coherence protocol. We set up the baseline cache hierarchy to resemble a SunFire 6800 server with 16 processors. The server uses UltraSPARC III processors, each equipped with two levels of caches. The processors have two separate first level caches, a 4-way associative 32 KB instruction cache and a 4-way associative 64 KB data cache. The second level cache is a 2-way associative cache of 8 MB, which is shared between data and instruction accesses.

We focus on the behavior of the second level caches and the interconnect in the experiments. All cache miss figures are for the second level cache and the traffic generated on the interconnect between the processors. The only hardware parameter that is varied is the cache line size, which is normally 64 B in the SunFire 6800 second level caches. The second level caches of this computer system are subblocked. To isolate the effects of a varying cache line size and to avoid corner cases in the prefetch experiments, the figures presented are for simulated non-subblocked second level caches. A comparative study was performed also with subblocked second level caches with a small increase in cache misses as a consequence.

The article only presents results for the measured cache misses and the traffic produced rather than to simulate the contention that may arise on the interconnect. This will not allow us to estimate the wall clock time for the execution of the benchmarks. Execution time is difficult to estimate based on simulation. If the memory access time is large, the execution time is highly dependent on the amount of cache misses. For communication intense applications, the execution is also very dependent on the contention on the interconnect. However, a study of the snoop-based Sun E6000 server showed that memory access time only is influenced by contention if the interconnect is overloaded [23]. The estimated execution time would therefore be highly implementation dependent on the bandwidth assumptions for the memory hierarchy and the bandwidth of coherence broadcast and data switches.

The characterization of cache misses in this article is influenced by Eggers and Jeremiassen [10]: The first reference to a given cache line by a processor is a cold miss. Subsequent misses to the same cache line by the same processor are either caused by invalidations and/or replacements. All misses caused by replacements are classified as capacity misses. The invalidation misses are either classified as false or true sharing misses. False sharing misses occur if another word in the cache line has been modified by another processor during the lifetime in the cache. All other invalidation misses are true sharing misses. Conflict misses are included in the capacity miss category.

4 Simulation Working Sets

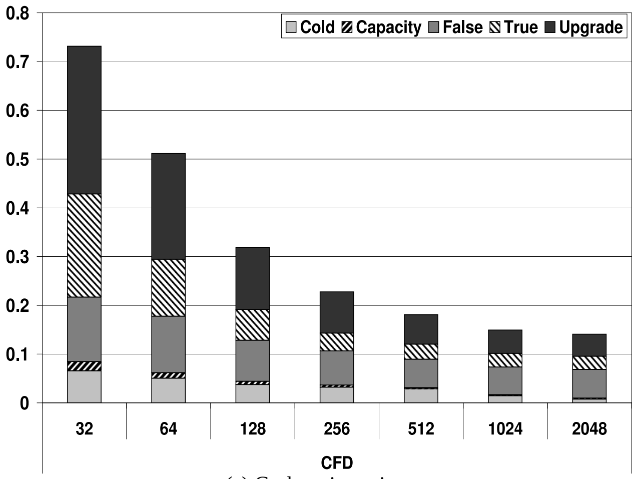
The CFD problem size has a grid size of $32 \times 32 \times 32$ elements using four multigrid levels. The CEM problem represents a modeled generic aircraft with a problem coefficient matrix of about $175{,}000 \times 175{,}000$ elements; a little more than 300,000 of these are non-zero. The QD problem size is $256 \times 256 \times 20$, i.e. a 256×256 2D FFT in the x-y directions followed by a 20 element FDM in the z-direction, a realistic size for problems of this type.

We only perform a number of iterative steps for each PDE kernel to limit the simulation time. Also a limited number of iterations gives appropriate results for the cache behavior since the access pattern is most often regular within each iteration for the PDE kernels. We performed a number of experiments with a larger number of iterations to verify that the results obtained were valid also for a shorter simulation. Before starting the measurements each solver completes a full iteration to warm-up the caches. The CFD solver runs for two iterations, the CEM solver runs until convergence (about 20 iterations) and the QD solver runs for three iterations.

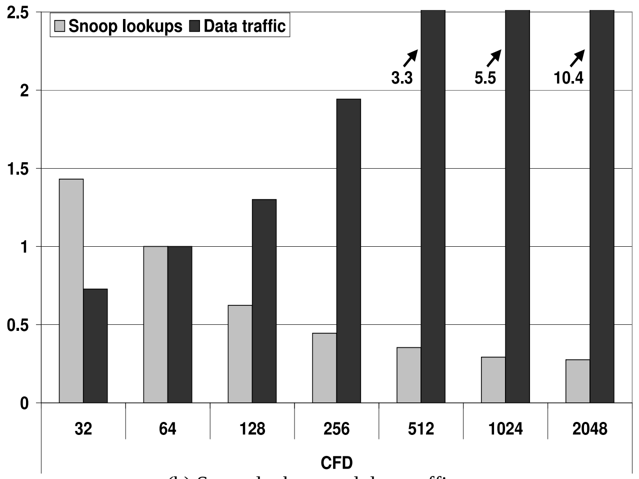
The PDE solvers are compiled using the Sun Forte developer 7 compiler with the *-fast* optimization flag set.

5 Impact of Varying Cache Line Size on the PDE Solvers

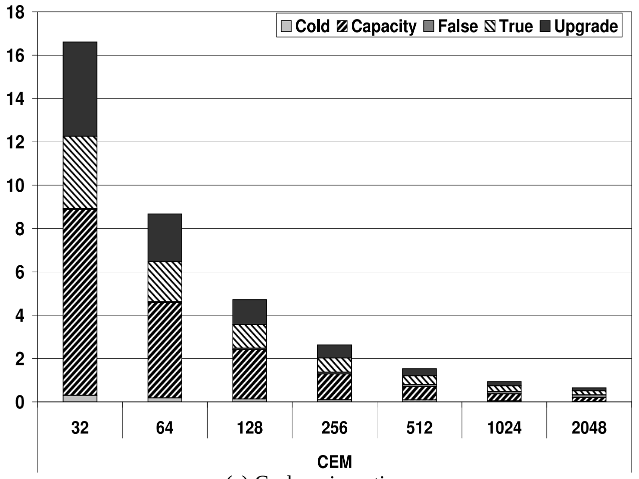
The cache miss characteristics, the required number of snoop lookups and the data traffic have been measured for different cache line sizes for the three PDE solvers in Figure 1. The cache misses are categorized into five different cache miss types according to the scheme described in Section 3. The *data traffic* is a measurement of the number of bytes transferred on the interconnect while the term *snoop lookups* represents the number of snoop lockups required



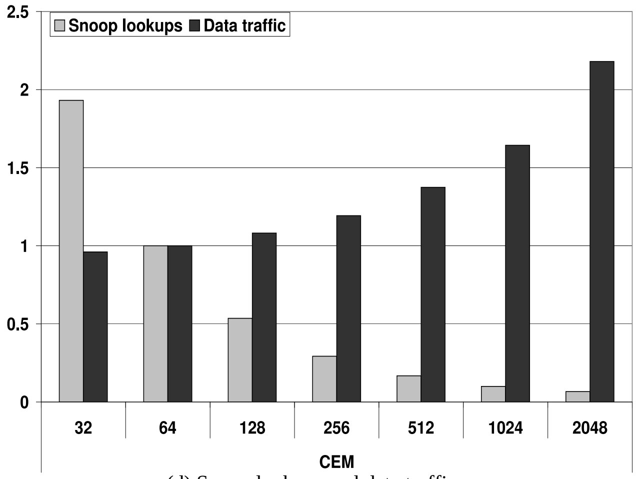
(a) Cache miss ratio



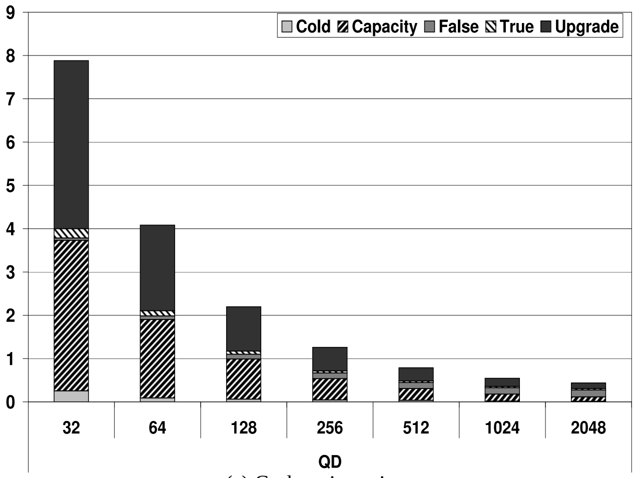
(b) Snoop lookups and data traffic



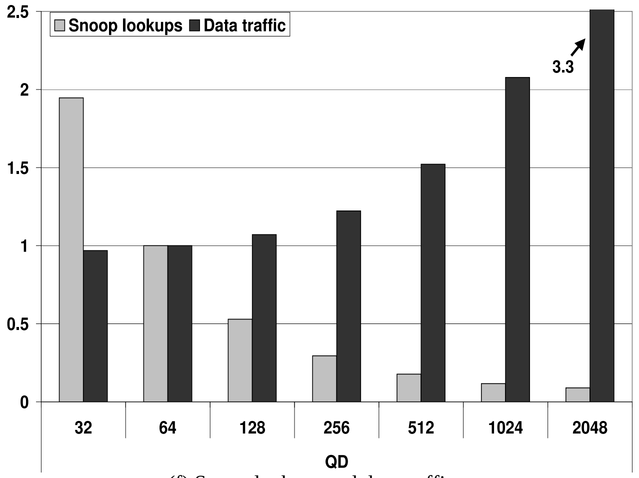
(c) Cache miss ratio



(d) Snoop lookups and data traffic



(e) Cache miss ratio



(f) Snoop lookups and data traffic

Figure 1. Influence of cache line size on cache misses, snoop lookups and data traffic in the three PDE kernels. The miss ratio in percent is indicated in the cache miss figures. The snoop lookups and the data traffic are normalized to 1.0 relative to the 64 B configuration.

by the caches on the interconnect. The snoop lookups is normally related to the number of cache misses that occurs in the system, while the data traffic normally increases with larger cache line size since larger amounts of data are transferred in each packet.

5.1 The CFD kernel

Most of the computations are carried out within each cell in the grid, leading to a low overall miss ratio in the CFD kernel. The decrease in miss ratio is substantial when the line size is increased, although somewhat smaller in comparison with the other solvers. The data causing the true sharing misses and the upgrades exhibit good spatial locality and the amount of these misses is halved with each doubling of the line size.

The decrease in cache miss ratio is influenced by a remarkable property in this kernel: the false sharing misses are reduced when the line size is increased. The behavior of false sharing is normally the opposite; false sharing misses increase with larger line size. When a processor is about to finish a smoothing operation it requests data that have previously been modified by another processor, i.e. all data have been invalidated. The invalidated data are placed consecutively in the remote cache. Larger pieces of this data are brought into the local cache when a longer cache line is used. If a shorter cache line is used, less invalidated data will be brought to the local cache in each access and therefore a larger number of accesses is required to bring all the requested data to the cache. All these accesses will be categorized as false sharing misses, thus generating more false sharing misses for shorter cache line sizes.

The miss ratio decreases slower at larger line sizes because the true and false sharing misses cannot be reduced below a certain threshold. This makes the amount of data traffic scale less well for long cache line sizes. The decrease in snoop lookups is approximately proportional to the decrease in miss ratio. Due to the large increase in data traffic, the ideal cache line size is shorter in this kernel than in the other kernels. However, the total miss ratio is also much lower.

5.2 The CEM kernel

The CEM kernel has a large problem size, which is clearly shown in the high miss ratio for small cache line sizes. Capacity misses are most common and can be avoided using a large cache line size. The true sharing misses and the upgrades are also reduced with a larger cache line size, but at a slower rate since the data vector is being randomly accessed. False sharing is not a problem in this application, not even for very large cache line sizes.

The data traffic exhibits nice properties for large cache lines because of the rapid decrease in the miss ratio. The address snoops are almost halved with each increase in cache line size. Therefore, the optimal cache line size for the CEM kernel is very long.

5.3 The QD kernel

This application is heavily dominated by two miss types, capacity misses and upgrades, which both decrease when the line size is increased. The large number of upgrades is caused by the all-to-all communication pattern during the transpose operation where every element is modified after being read. This should result in an equal number of true sharing misses and upgrades, but the kernel problem size is very large and replacements take place before the true sharing misses occur. Therefore, a roughly equal amount of capacity misses and upgrades are recorded in the model.

The bus traffic increases rather slowly for large cache lines. It is only at cache line sizes larger than about 256 B that the data traffic begins to increase at a faster rate. The snoop lookups has the opposite behavior and decreases rapidly until the 256 B cache line size, where it levels out.

6 Sequential Hardware Prefetching

The results presented in Section 5 showed that a very long cache line would be preferable in a computer optimized for solving PDEs. A preferable cache line size would probably be between 256 and 512 B for these applications to make better use of spatial locality. Even larger cache lines lead to less efficiently used caches with a large increase in data traffic as a consequence. Unfortunately, no computer is built with such large line size. Instead, a similar behavior to having a large cache line can be obtained by using various prefetch techniques. These techniques try to reduce the miss penalty by prefetching data to the cache before the data are being used. The proposed schemes are either software-based [19, 20, 26] or hardware-based [3, 6, 7, 12, 14, 17, 24].

A very simple hardware method to achieve cache characteristics similar to having a large cache line, while keeping a short cache line size, is to implement sequential prefetching. In such systems, a number of prefetches is issued for data having consecutive addresses each time a cache miss occurs. The amount of prefetching is governed by the prefetch degree, which decides how many prefetches to perform on each cache miss. Dahlgren et al. has studied the behavior of sequential prefetching on the SPLASH benchmarks [6]. Sequential prefetching normally efficiently reduces the number of cache misses since more data is brought into the cache on each cache miss. Sequential prefetching requires only small changes to the cache controller and can easily be implemented without a large increase in coherence complexity. The main disadvantage with sequential prefetching schemes is that the data traffic and the address snoops usually increase heavily.

The snoop lookups can be largely reduced in sequential prefetching using the bundling technique presented in a previous publication [28]. Bundling lumps the original read request together with the prefetch requests to the consecutive addresses. The original read request is extended with a bit mask, which shows the address offset to the prefetch addresses. Bundling efficiently limits the number of address transactions on the interconnect. However, not only the address traffic can be reduced but also the number of snoop lookups performed. This is done by requiring bundled read prefetch requests to only supply data if the requested cache line is in the owner state in the remote cache. Data that are in other states will not be supplied to the requesting cache. Write and upgrade prefetches are not bundled and will not reduce the amount of generated address traffic or the required snoop lookups. Read bundling gives a large performance advantage since the available snoop bandwidth usually is the main contention bottleneck in snoop-based multiprocessors. The available data bandwidth is a smaller problem since the data packets do not have to be ordered and can be returned on a separate network such a point-to-point network or a crossbar switch. For example, the Sunfire 6800 server has a data interconnect capable of transferring 14.4 GB/s while its snooping address network can support 9.6 GB/s worth of address snoops [4].

7 Impact of Sequential and Bundled Sequential Prefetching on the PDE solvers

We have studied the characteristics of sequential hardware prefetching on the PDE solvers. The results can be seen in Figure 2. A table describing the tested configurations is given in Table 1. In the figure, we present results of non-prefetching configurations having different cache line sizes: *64*, *128*, *256* and *512* B. Several sequential prefetching configurations, *64s1*, *64s3* and *64s7*, are also studied which prefetch 1, 3 and 7 consecutive cache lines based on address on each cache miss. Finally the corresponding bundled configurations prefetching 1, 3 and 7 consecutive addresses, *64b1*, *64b3*and *64b7*, are evaluated.

As can be seen in Figure 2, sequential prefetching works very well with the studied PDE solvers. If we compare the sequentially prefetching configurations, *64s1*, *64s3* and *64s7*, with the non-prefetching configuration, *64*, we see that for all kernels the cache misses are largely reduced. Compared with the non-prefetching configuration with a comparable cache line size, e.g the *512* B configuration compared to the *64s7* configuration, the cache misses are lower or equal for the prefetching configuration. The

config	pref. degree	bundled
64	0	-
64s1	1	no
64b1	1	yes
128	0	-
64s3	3	no
64b3	3	yes
256	0	-
64s7	7	no
64b7	7	yes
512	0	-

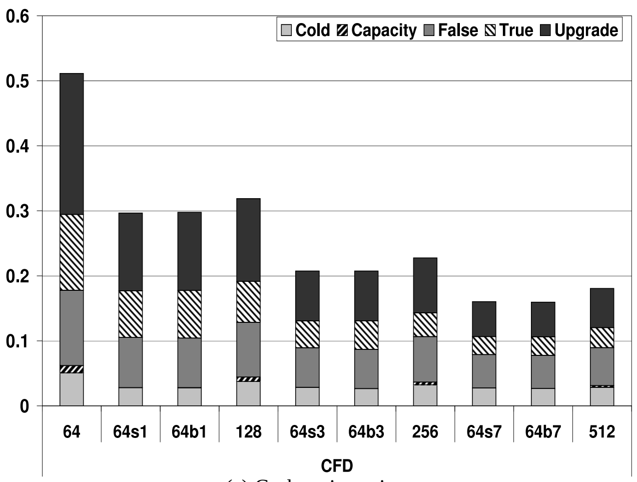
Table 1. Configurations

main reason for the discrepancy is that with the sequential prefetching protocol, the consecutive addresses will always be fetched. If a protocol with a large cache line is used, a cache line aligned area around the prefetched address will be fetched, that is, both data before and after the desired address can be fetched. Especially the CEM kernel takes large advantage of this, where the cache misses are reduced about 30 percent in the sequential protocol compared to the large cache line non-prefetching protocol.

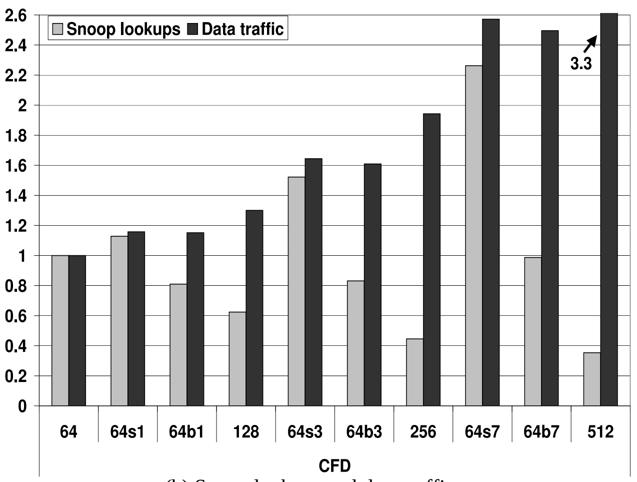
The data traffic increases with enlarged cache line size as was previously shown. Sequential prefetching generally leads to more data traffic than the baseline *64* B non-prefetching configuration and less data traffic than the corresponding larger non-prefetching configuration. The snoop lookups on the other hand increase rather heavily for the CFD and QD kernels compared with both a small and a large cache line size non-prefetching configuration.

Bundling can be used to efficiently reduce the overhead in address snoops. For all kernels, bundling yields an almost equal amount of cache misses as the corresponding sequential prefetch protocol. However, both the snoop lookups and the data traffic are reduced in the bundled protocols for all kernels. The reason for the decrease in data traffic is that the bundled protocol is more restrictive at providing data than the sequential protocol. Data are only sent if the owner of the original cache line is also the owner of the prefetch cache lines. Bundling makes it possible to use a large amount of prefetching with a much smaller cost, especially in address snoops, than normal sequential prefetching would have. There is still more snoop lookups generated in the bundled prefetch protocol than in a non-prefetching large cache line size protocol, which is almost entirely caused by upgrades generating prefetch messages. This effect cannot be eliminated since only reads are bundled.

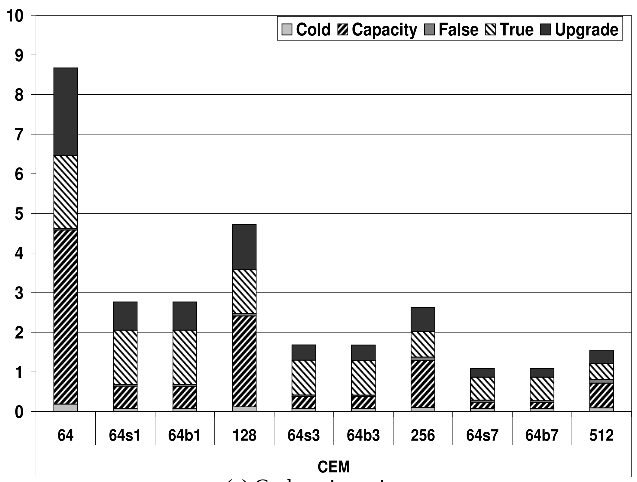
The overall performance for the bundled sequential protocols compared with the baseline 64 B non-prefetching protocol is excellent especially for the CEM and QD-kernels. For example the *64b7* configuration has about 88



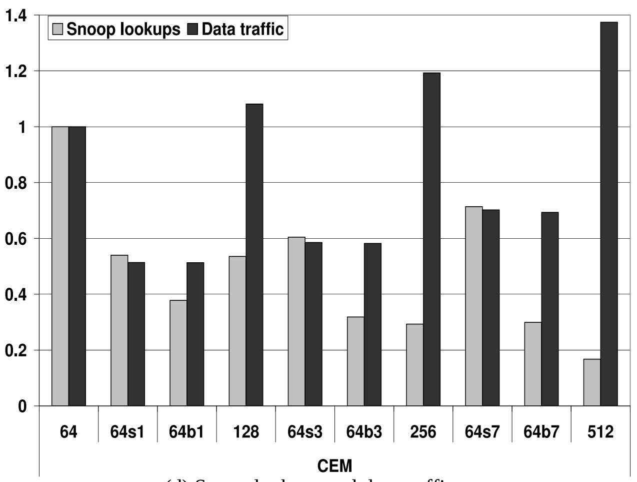
(a) Cache miss ratio



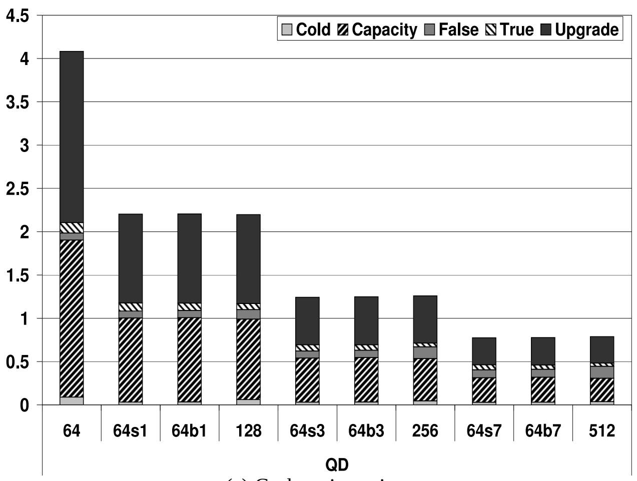
(b) Snoop lookups and data traffic



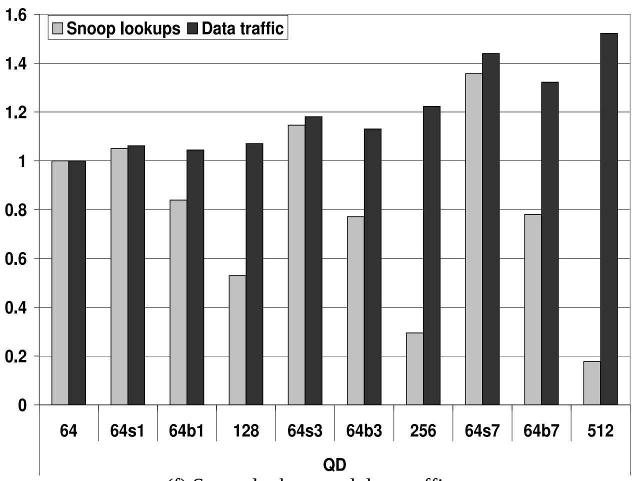
(c) Cache miss ratio



(d) Snoop lookups and data traffic



(e) Cache miss ratio



(f) Snoop lookups and data traffic

Figure 2. Influence of sequential and bundled sequential prefetching on cache misses, snoop lookups and data traffic. The miss ratio in percent is indicated in the cache miss figures. The snoop lookups and the data traffic are normalized to 1.0 relative to the 64 B configuration.

percent less cache misses, 70 percent less snoop lookups and 30 percent less data traffic than the 64 B non-prefetching protocol for the CEM kernel. The performance of the bundled protocols in the CFD kernel is somewhat worse since the data traffic increases rather heavily compared with the non-prefetching protocol. However, since the total ratio of cache misses is small in this kernel, contention on the interconnect will most likely not be a performance bottleneck. Also, the snoop bandwidth is often more limited than data bandwidth in snoop-based systems.

Compared with previous studies [28], the PDE kernels gain more from read bundling than the SPLASH-2 benchmarks and commercial JAVA-servers. This is an effect of more carefully coded applications, which make better use of spatial locality.

Sequential prefetching can simply be implemented in a multiprocessor without a large increase in hardware complexity. Some extra hardware is needed in the cache controller to fetch more data on each cache miss. The cost of implementing a bundled protocol is rather small even though some extra corner cases can be introduced in the cache coherence protocol [28].

8 Conclusion

From full-system simulation of realistically sized state-of-the-art PDE solvers we have learned that these applications do not experience problems with false sharing as is the case in many benchmark programs and commercial applications. Also, the spatial locality is good in the PDE solvers and therefore it would be beneficial to use computers with large cache line sizes for solving these problems.

Since most commercial computers are built with rather short cache lines, a simple hardware sequential prefetching scheme, can be used to yield a similar behavior to having a large cache line size. The performance of the studied PDE solvers takes advantage of sequential prefetching.

To further improve the performance of the PDE solvers, sequential prefetching can be used together with the bundling technique to largely reduce the amount of address snoops required by the caches in shared-memory multiprocessors. Using this technique, both the number of cache misses and the required snoop lookups become lower than in a non-prefetching configuration.

Acknowledgement

We would like to thank Jim Nilsson for providing us with the original version of the cache coherence protocol model for Simics. We also would like to express gratitude to Markus Nordén for the CFD kernel and to Henrik Löf for the CEM kernel modeled in this paper.

This work is supported in part by Sun Microsystems, Inc. and the Parallel and Scientific Computing Institute (PSCI), Sweden.

References

[1] http://www.spec.org/.

[2] http://www.tpc.org/.

[3] J.-L. Baer and T.-F. Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991 Conference on Supercomputing*, pages 176–186, 1991.

[4] A. Charlesworth. The Sun Fireplane System Interconnect. In *Proceedings of the 2001 Conference on Supercomputing*, 2001.

[5] A. Chronopoulos and C. Gear. S-Step Iterative Methods for Symmetric Linear Systems. *Journal of Computational and Applied Mathematics*, 25:153–168, 1989.

[6] F. Dahlgren, M. Dubois, and P. Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, 1995.

[7] F. Dahlgren and P. Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):385–398, 1996.

[8] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 88–97, 1993.

[9] F. Edelvik. *Hybrid Solvers for the Maxwell Equations in Time-Domain*. PhD thesis, Department of Information Technology, Uppsala University, 2002.

[10] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 377–381, 1991.

[11] S. J. Eggers and R. H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, 1989.

[12] E. H. Gornish. *Adaptive and Integrated Data Cache Prefetching for Shared-Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[13] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, 1992.

[14] E. Hagersten. *Toward Scalable Cache-Only Memory Architectures*. PhD thesis, Royal Institute of Technology, Stockholm, 1992.

[15] A. Jameson and D. Caughey. How Many Steps are Required to Solve the Euler Equations of Steady, Compressible Flow: in Search of a Fast Solution Algorithm. In *Proceedings of the 15th Computational Fluid Dynamics Conference*, 2001.

[16] M. Karlsson, K. Moore, E. Hagersten, and D. A. Wood. Memory System Behavior of Java-Based Middleware. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, 2003.

[17] D. M. Koppelman. Neighborhood Prefetching on Multiprocessors Using Instruction History. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 123–132, 2000.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[19] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.

[20] T. C. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions on Computer Systems (TOCS)*, 16(1):55–92, 1998.

[21] M. Nord´en, M. Silva, S. Holmgren, M. Thun´e, and R. Wait. Implementation issues for high performance cfd. In *Proceedings of International Information Technology Conference, Colombo, Sri Lanka*, 2002.

[22] A. Petersson, H. Karlsson, and S. Holmgren. Predissociation of the Ar-12 van der Waals Molecule, a 3d Study Performed Using Parallel Computers. Submitted to Journal of Physical Chemistry, 2002.

[23] A. Singhal, D. Broniarchyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, and E. Hagersten. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of IEEE Hot Interconnects*, pages 41–52, 1996.

[24] M. K. Tcheun, H. Yoon, and S. R. Maeng. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the International Conference on Parallel Processing*, pages 306–313, 1997.

[25] J. Torrellas, M. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[26] D. M. Tullsen and S. J. Eggers. Effective Cache Prefetching on Bus-Based Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 13(1):57–88, 1995.

[27] D. Wallin. Performance of a High-Accuracy PDE Solver on a Self-Optimizing NUMA Architecture. Master's thesis, Department of Information Technology, Uppsala University, 2001.

[28] D. Wallin and E. Hagersten. Miss Penalty Reduction Using Bundled Capacity Prefetching in Multiprocessors. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.