Miss Penalty Reduction Using Bundled Capacity Prefetching in Multiprocessors

Dan Wallin and Erik Hagersten
Uppsala University
Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{danw, eh}@it.uu.se

Abstract

While prefetch has proven itself useful for reducing cache misses in multiprocessors, traffic is often increased due to extra unused prefetch data. Prefetching in multiprocessors can also increase the cache miss rate due to the false sharing caused by the larger pieces of data retrieved.

The capacity prefetching strategy proposed in this paper is built on the assumption that prefetching is most beneficial for reducing capacity and cold misses, but not communication misses. We propose a simple scheme for detecting the most frequent communication misses and suggest that prefetching should be avoided for those. We also suggest a simple and effective strategy for reducing the address traffic while retrieving many sequential cache lines called bundling.

In order to demonstrate the effectiveness of these approaches, we have evaluated both strategies for one of the simplest forms of prefetching, sequential prefetching. The two new strategies applied to this bandwidth-hungry prefetch technique result in a lower miss rate for all studied applications, while the average amount of address traffic is reduced compared with the same application run with no prefetching. The proposed strategies could also be applied to more sophisticated prefetching techniques for better overall performance.

1. Introduction

Building an optimized computer system is a minimization effort spanning over many properties. As a designer, you want to minimize the number of cache misses, number of address transactions and the data bandwidth consumed by a wide variety of applications, while keeping the implementation complexity reasonable.

An important first step in this optimization effort is to choose the key design parameters, such as cache line size, degree of subblocking, cache associativity, prefetch strategy, etc. The problem in finding an "optimum setting" for these design parameters is that, while improving one property, some other may become worse. For example, a slightly longer cache line often decreases the cache miss rate and address traffic, while the data bandwidth increases. Enlarging the cache line can also result in increased data traffic, as well as increased address traffic, since misses caused by false sharing may start to dominate. A further complication is that application behaviors differ greatly. A setting that works well for one application may work poorly for others.

Ideally, one would like to dynamically select the best strategy for each application. Better yet would be dynamically selecting the best strategy for each class of data within each application. While the complexity of such a proposal may be prohibitive, it would be compelling to at least select different strategies for handling the two largest classes of cache misses in an application: capacity misses and communication misses.

It is well known that large cache lines are often beneficial to data that cause capacity misses due to spatial locality. However, the access pattern of communication often does not take advantage of larger cache lines and may suffer from false sharing. We propose a simple method for distinguishing between communicating and non-communicating cache lines. Extra data are prefetched for non-communicating misses, but not for communicating misses. Prefetching normally introduces a lot of extra address traffic. By bundling the prefetching requests with the original request the address traffic can be largely reduced.

The proposed strategies cause the number of cache misses to decrease while keeping the data and address traffic under control compared with a non-prefetching protocol. In our study, we have applied the bandwidth-reducing strategies to a sequential prefetching protocol with a prefetch distance of 7 and shown that the strategies can improve the protocol significantly. The intention of this paper is not to recommend a sequential prefetch protocol as a final solution for prefetching, but rather to show the potential of separat-

ing prefetching for communicating and non-communicating data. A rather large prefetch distance has been chosen to clearly distinguish between prefetching for communicating and non-communicating data.

2. Background

Many studies looking for the optimum cache line size were performed in the early nineties. It was discovered that small cache lines increased the amount of capacity misses while too large cache lines could ruin the effectiveness of a cache. For caches in coherent shared-memory multiprocessors, this effect is even more pronounced. Here, a large cache line may not only make the cache less efficiently used, but will also increase the amount of false sharing. For each application, there is an optimum size resulting in the "best" performance [16]. Studies have been carried out for sequential hardware prefetching schemes, where the cache line size is kept small but a number of subsequent cache lines are prefetched on each cache miss to yield a behavior similar to a large cache line [8]. Also here, the "best" prefetch distance can be decided on a per-application basis, which also changes at run-time.

2.1. Choosing the Cache Line Size

In uniprocessors, a very simple and usually efficient method of decreasing the cache misses is to enlarge the cache line size [4], [14], [23]. However, the number of capacity misses increases because unnecessary data are brought into the cache for large cache line sizes. Torrellas et al. showed that larger cache lines in multiprocessors do not decrease the cache misses as efficiently as in uniprocessors due to false sharing and poor spatial locality in shared data [25]. The optimum cache line size is therefore usually smaller for multiprocessors than for uniprocessors.

Several papers have investigated the effects of cache line size on miss rate and traffic in multiprocessors [10], [12], [16], [27]. In Figure 1, we have carried out a similar cache miss and traffic analysis for the four SPLASH- 2 kernel applications [27] on 16 processors, each with 1 MB of cache. Also the miss ratio, indicating the percentage of cache misses of all cache accesses can be found in the figure. The U-shaped curve is especially pronounced in the Radix-application. The number of false misses increases with large cache line size due to the false sharing in all applications.

2.2. Prefetching in Multiprocessors

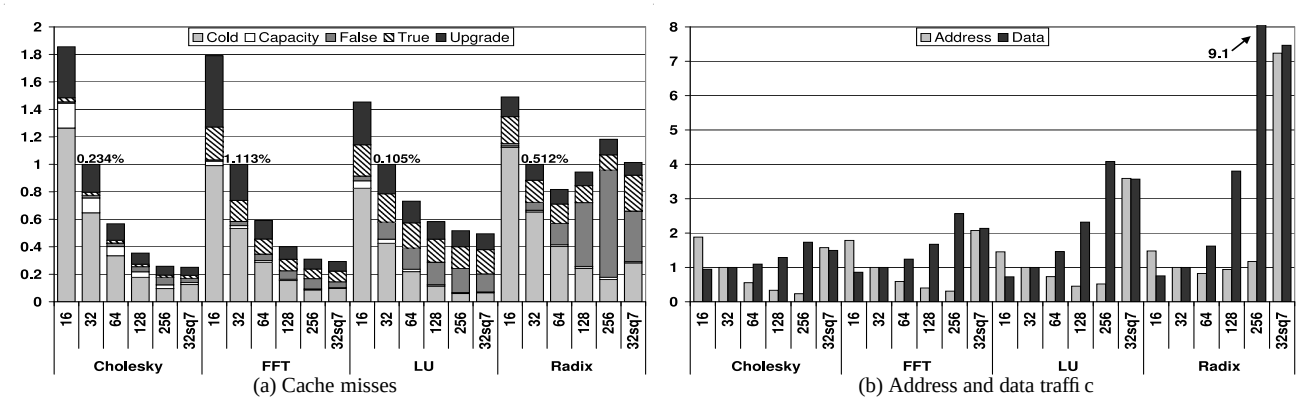
Several researchers have studied prefetching in multiprocessors as a method of reducing the miss penalty. The proposed prefetching schemes are either software-based [21], [22], [26] or hardware-based [5], [8], [9], [15], [17], [19], [24]. While software approaches often introduce an instruction overhead, hardware approaches often lead to increased memory traffic. Chen and Baer [7] therefore propose using a combination of hardware and software prefetching.

Hardware approaches to prefetching in multiprocessors are usually based on either stride prefetching or sequential prefetching. While sequential prefetching prefetches the consecutive addresses on a cache miss, stride prefetching prefetches addresses a certain distance away from the previous cache miss. Stride prefetching has a certain learning time during which the prefetcher computes which address to prefetch next. The efficiency of sequential and stride prefetching depends largely on the access pattern behavior [13]. Dahlgren and Stenström showed that sequential prefetching normally reduces cache misses more than stride prefetching. This is caused by the learning time and that most strides fit into a single cache line [9]. However, stride prefetching protocols often consume less memory bandwidth than sequential prefetch protocols since they in general are more restrictive about issuing prefetches.

Baer and Chen [5] proposed to predict the instruction stream with a look-ahead program counter. A cache-like reference predictor table is used to keep previous predictions of instructions. Correct branch prediction is needed for successful prefetching. Several authors have studied the behavior of adaptive protocols in multiprocessors that vary the degree of prefetching at run-time. Dahlgren et al. presented an implementation which detects whether prefetched data has been used or not by tagging cache lines and varies the degree of prefetching on the success of previous prefetches [8]. Other adaptive protocols introduce small caches which detect the efficiency of prefetches based on the data structure accessed [15], [17], [24]. These approaches require complex hardware and have problems detecting irregular strides. Koppelman presented a prefetching scheme that can be seen as a compromise of stride and sequential prefetching called neighborhood prefetching [19]. It uses the instruction history to compute an area around the demanded data, which can be prefetched.

2.3. Effects of Sequential Prefetching

The behavior of sequential prefetching for the SPLASH-2 kernel benchmarks running on a 16 processor system with one level of 1 MB caches can be studied in Figure 1. For the sequential prefetching configuration, the cache line size is 32 B. On each cache miss the consecutive 7 cache lines are prefetched. As can be seen, the miss rate is substantially reduced in three of the four applications, while both the address and data traffic increase compared with the 32 B non-prefetching scheme. The sequential prefetching configuration has a similar miss rate to the protocol with a 256 B



(a) Cache misses (b) Address and data traffi c

Figure 1. Influence of cache line size on the kernel SPLASH-2 applications. The simulations are run on 16 processors, each with 1 MB 4-way associative caches. The *32sq7* configuration uses a cache line size of 32 B and sequentially prefetches the consecutive 7 lines on each cache miss. The cache misses, address traffic and data traffic are normalized relative to the *32* B configuration. The miss ratio for each application is indicated for the *32* B configuration.

cache line for most applications, since enlarging the cache line size is basically a simple form of prefetching. However, a much higher address bandwidth is required.

For the Radix application, the miss rate is somewhat lower for the sequential prefetch scheme than for the 256 B configuration. The reason for the lower cache miss rate is that, with a longer cache line size, only a single processor can reply to requests for the entire 256 B of data. Also, the sequential prefetch scheme always prefetches subsequent blocks on a cache miss. The data traffic drops somewhat because prefetch requests are only generated if the cache line is not already available in the cache.

3. Miss Penalty Reducing Prefetching Schemes

Sequential prefetching schemes introduce a large amount of data and address traffic compared with a non-prefetching coherence protocol. We propose two new schemes for lowering the bandwidth requirements while taking advantage of a low cache miss rate: *capacity prefetching* and *bundled capacity prefetching*.

3.1. Capacity Prefetching

One problem with previous prefetching schemes in multiprocessors is the increase in false sharing misses for large cache line sizes. Would it be possible to distinguish between lines causing false sharing and other cache lines? The idea of the capacity prefetching algorithm is to prefetch only if the cache line causing the miss has been replaced from the cache or is being upgraded from the *Shared* to the *Modified* state. These kinds of cache misses can be identified without the need for any more state information in the cache. We simply distinguish between cache lines present in the cache in the *Invalid* state and cache lines not present in the cache (which are also interpreted as *Invalid* by normal cache coherence protocols).

Since non-communicating cache lines do not cause any false sharing misses, it should be beneficial to make them longer to take better advantage of spatial locality. However, communicating lines are more likely to be involved in false sharing situations and should be kept shorter. On a cache miss, two possible actions can be taken in capacity prefetching. Either the cache line has been replaced or has never been accessed before, in which case prefetching will be performed, or the cache line has been invalidated and only the missing cache line will be fetched into the cache. So, in capacity prefetching, the actual cache line size will effectively appear longer for non-communicating lines.

3.2. Bundled Capacity Prefetching

For sequential prefetching, not only will the cache line that caused the miss be fetched into cache, but a number of other cache lines will as well. The static prefetching distance decides the number of lines to fetch on each miss. The prefetched lines are brought into the cache or upgraded by generating another *Read*, *ReadExclusive* or *Upgrade* request on the bus for each prefetch. Obviously, this introduces a lot of extra address traffic, especially for large prefetch distances.

The extra address traffic caused by the prefetches could be largely reduced by bundling the original *Read*, *ReadExclusive* and *Upgrade* request together with the prefetch requests. An easy way to do this is to extend the requests with a bitmap indicating which lines beyond the original request to prefetch to the cache. However, while this would reduce

the number of address transactions on the bus, it would not reduce the number of snoop lookups each cache has to perform. Also, it may create a multisource situation, where a single address transaction would result in data packets being transferred from many different sources. This would violate some of the basic assumptions of state-of-the-art implementation of cache coherence [6].

Therefore, we propose a more restrictive approach to the bundling principle. The first limitation is not to bundle *ReadExclusive* requests. Since these transactions may cause changes to any of the snooping caches, they would still require a snoop lookup in each cache for each bundled cache line. The second limitation is that only the owner of the originally requested line will supply data on a *Read* prefetch request. The owner will only reply with data if it is also the owner of the requested prefetch line. This way, *only* the owner may have to snoop the bundled cache lines. As a consequence not only will the address traffic decrease for *Read* prefetches but also the number of snoop lookups.

4. Simulation Environment

All experiments were carried out using execution-driven simulation in the full-system simulator Simics modeling the SPARC v9 ISA [20]. An invalidation-based MOSI cache coherence protocol extension to Simics was used. In all experiments, a bus-based 16 processor system with one level of 1 MB 4-way associative unified data and instruction caches per CPU were modeled. Since our goal is to reduce the level 2 cache misses and we assume an inclusive cache hierarchy, we have chosen to model only one cache level. We have further selected to only report the traffic produced at different levels rather than simulatíng the contention that may arise from the traffic. While this will not allow us to estimate the wall clock time for the execution of the benchmarks, which would be highly implementation dependent, it will isolate the parameters we are trying to minimize.

Many different classification schemes for cache misses in multiprocessors have been proposed [10], [11], [25]. The cache miss characterization in our paper is influenced by Eggers and Jeremiassen [11]: The first reference to a given block by a processor is a cold miss. Subsequent misses to the same block 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 its lifetime in the cache. All other invalidation misses are true sharing misses. Conflict misses are included in the capacity miss category.

4.1. Benchmark Programs and Working Sets

The studies have been carried out on the SPLASH- 2 programs [27] and two commercial workloads, SPECjbb2000 [3] and ECperf [1] (a slightly modified version has lately been adapted as SPECjAppServer2001 [2]). The SPLASH-2 programs are mainly scientific, engineering, and graphics applications and have been used extensively as multiprocessor benchmarks the last few years. However, SPLASH-2 does not provide realistic results for computer designers since the server market is entirely dominated by commercial applications such as databases and application servers.

ECperf and SPECjbb2000 are both Java-based middleware benchmarks. ECperf is a benchmark modeling Java Enterprise Application Servers that uses a number of Java 2 Enterprise Edition (J2EE) APIs in a web application. ECperf is a complicated multi-tier benchmark that runs on top of a database server and an application server. SPECjbb2000 evaluates the performance of server-side Java. It can be run on any Java Virtual Machine. Both are commercial benchmarks that put heavy demands on the memory and cache system. More information on SPECjbb2000 and ECperf can be found in [18].

The SPLASH-2 workloads have been chosen according to the default values specified in the SPLASH-2 release [27] with some minor changes: the Cholesky benchmark was optimized for 1 MB caches, the FFT benchmark was run with 65536 data points, the Raytrace benchmark allocated a total of 64 MB global memory, and the Radiosity benchmark used the small test scene provided in the distribution instead of the default room scene in order to limit the simulation time. All benchmarks were run using 16 parallel threads, and the measurements were started right after the child processes had been created in all applications except Barnes and Ocean, where the measurements were started after two time steps.

ECperf models the number of successfully completed "benchmark business operations" during a time period. The operations include business transactions such as a customer making an order, updating an order or checking the status of an order. The ECperf transactions take long a time, and a total of 10 transactions were run with a 3- transaction warm-up period. SPECjbb2000 transactions take much less time, and we simulated 50,000 transactions, including 10,000 transactions of warm-up time.

5. Experimental Results

The efficiency of capacity prefetching is presented in Figure 2. In the experiments, the cache line size is set to 32 B for all prefetching configurations. As a comparison, miss rate and bus traffic are also presented for the basic 32 B and 256 B cache line configurations without prefetch-

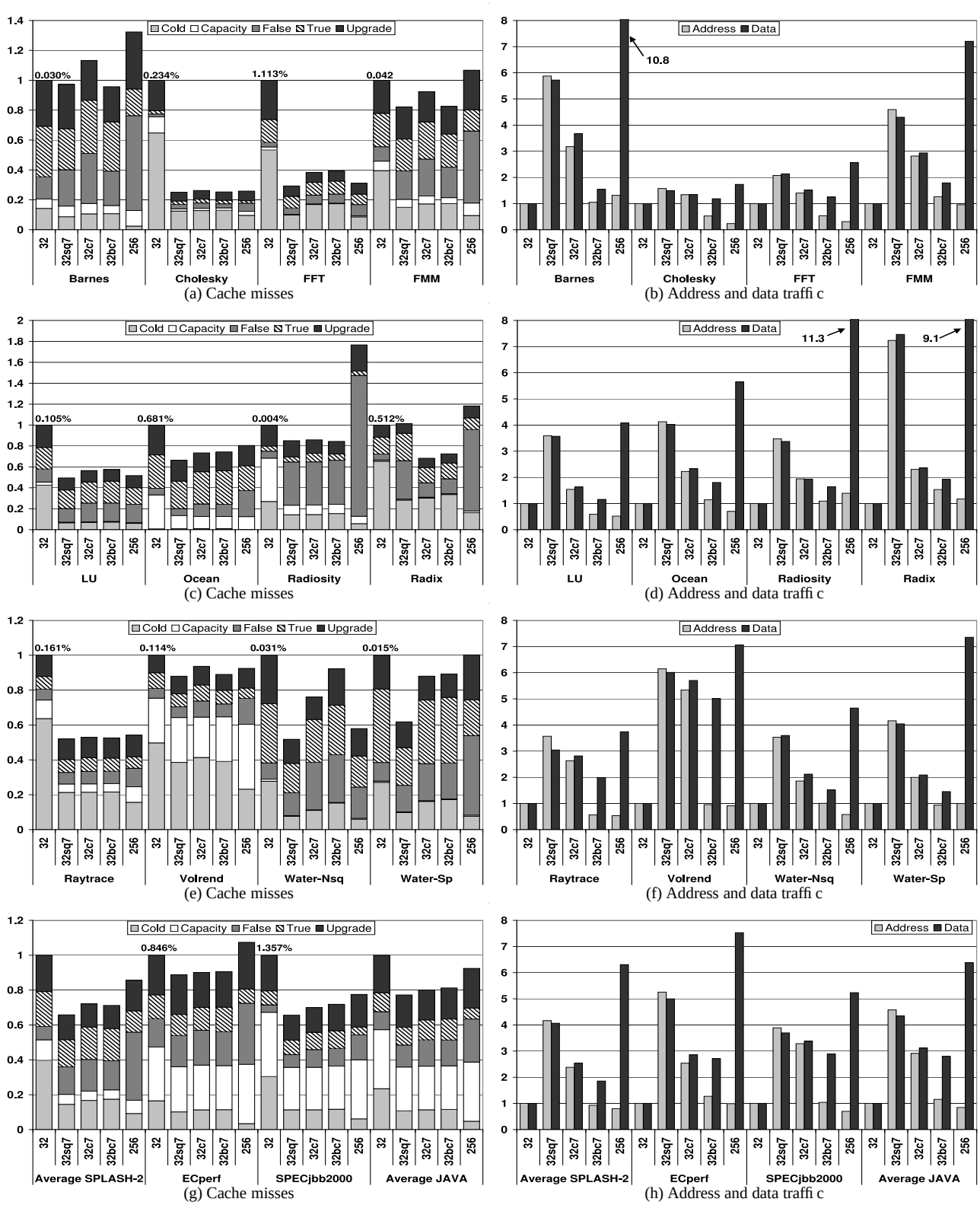


Figure 2. Effect of capacity prefetching in the SPLASH-2 and JAVA-server benchmarks. The non-prefetching protocols have cache line sizes of *32* and *256* B. The prefetching protocols include the sequential prefetch protocol, *32sq7*, the capacity prefetch protocol, *32c7*, and the bundled capacity prefetch protocol, *32bc7*. The cache misses, address traffic and data traffic are normalized relative to the *32* B configuration. The miss ratio for each application is indicated for the *32* B configuration.

ing. The results are presented for sequential prefetching, capacity prefetching and bundled capacity prefetching, all with a prefetch distance of 7 cache lines. The reason for prefetching 7 consecutive cache lines is to get an actual behavior similar to a long 256 B cache line and at the same time keep a short baseline cache line size.

A comparison between capacity prefetching and the sequential prefetching scheme shows that address and data traffic is reduced for capacity prefetching in all applications. The data and address traffic is further reduced using bundled capacity prefetching for all applications. Bundled capacity prefetching is especially efficient in traffic reduction, which is not surprising, since the method combines several bus requests. The reason for the reduction of the data traffic is that only the owner of the original cache line replies to the prefetch requests. Bundled capacity prefetching reduces the data traffic with about 50 percent and the address traffic by 75 percent compared with the sequential prefetching scheme as an average of all studied applications.

The data traffic is reduced in all prefetching schemes compared with a 256 B cache line size. Compared with the 256 B configuration, bundled capacity prefetching reduces the data traffic by about 70 percent on the average for all SPLASH- 2 benchmarks and about 55 percent for the JAVA-servers. On the average, the address traffic increases a few percent for bundled capacity prefetching compared with a 256 B cache line size and is similar to the 32 B cache line address traffic.

All prefetching schemes, except capacity prefetching in Barnes, manage to reduce the number of cache misses compared with the non-prefetching 32 B version for each application. Only three of a total of fourteen benchmarks show a worse miss rate for the bundled capacity prefetching scheme than the 256 B scheme: FFT, LU and Water-Nsq.

The average miss statistics from both the SPLASH- 2 benchmarks and the JAVA-server benchmarks show that sequential prefetching reduces the miss rate most. The difference in misses between capacity prefetching and bundled capacity prefetching is very small on the average and is caused by the owner reply limitation described in Section 3.2. Some applications, e.g., Barnes, FMM, Radix and Water-Nsq, show rather large differences in cache miss rates between the three prefetching configurations. In Barnes and FMM, bundled capacity prefetching is more efficient than capacity prefetching in two ways, by reducing cache misses and traffic. This is because no *ReadExclusive* prefetches are generated by *Upgrade* requests in the bundling, which hurts the miss rate in these applications.

Capacity prefetching and bundled capacity prefetching manage to reduce the number of capacity and cold misses (non-communication misses) while keeping the upgrades, false and true sharing misses (communication misses) under control. The applications with the largest false sharing problems for large cache line sizes (Barnes, FMM, Radiosity, Radix, Water-Sp, ECperf) gain the most by capacity prefetching. Some applications (Barnes, Radiosity, Water-Nsq, Water-Sp) introduce more false sharing when prefetching than could be found in the 32 B configuration. However, these applications experience even more false sharing misses for the 256 B non-prefetching configuration.

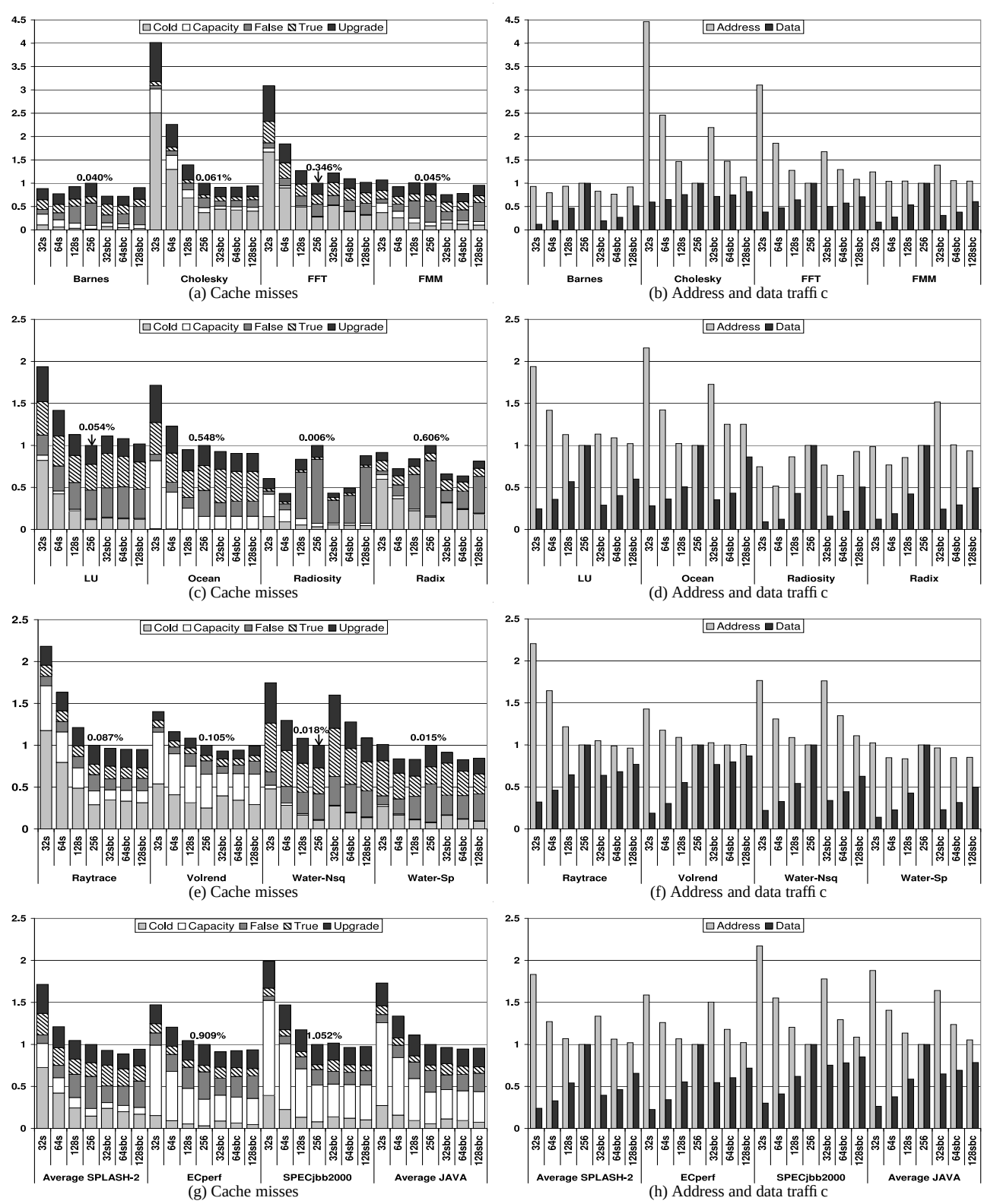
As can be seen in the applications with a large amount of capacity misses (Specjbb2000, ECperf, Volrend), the data traffic is not reduced as much as in applications with few capacity misses. This is not surprising since the prefetching takes place when a line is replaced in the cache. Despite this fact, the data traffic is reduced significantly compared with the 256 B cache line configurations.

Capacity prefetching only distinguishes between communication and non-communication cache lines. No distinction is made between false and true sharing misses. The prefetching is therefore suppressed not only for capacity misses but also for true sharing misses. However, for most of the studied benchmarks, this does not seem to increase the number of cache misses compared with the sequential prefetching protocol, which prefetches on all cache misses. Water-Nsq is the benchmark that is most influenced by this lack of prefetching. The application evaluates forces and potentials in water molecules. Each molecule has a record of data saved in an array. The size of the record is larger than 32 B. When a new processor updates the record, the whole record has to be transferred. Thus, in this application it would be advantageous to communicate large chunks of data, which is not the case in capacity prefetching. Since communication misses are handled the same way as in a non-prefetching protocol, the capacity prefetching protocols perform equally to the 32 B non-prefetching protocol.

6. Implementation Issues with Capacity Prefetching

The results presented in the former section showed that, with the bundled capacity prefetching protocol, it would be beneficial to reduce the cache line size for communicating lines and use a rather large prefetch distance for non-communicating lines. Shorter cache lines introduce a higher memory overhead in the cache implementation. A common way to reduce the overhead is to use subblocked caches. In a subblocked cache, a single address tag is associated with several cache lines, while each cache line has its own state tag. Subblocked caches yield more capacity misses than non-subblocked caches since the number of replacements increases with a less efficient use of the cache. A sounder comparison would therefore be to compare the bundled capacity prefetch protocol with cache implementations having the same cache tag size.

With a subblocked version of bundled capacity prefetch-



(a) Cache misses (b) Address and data traffic

(c) Cache misses (d) Address and data traffic

(e) Cache misses (f) Address and data traffic

(g) Cache misses (h) Address and data traffic

Figure 3. Effect of subblocked bundled capacity prefetching. All configurations have an address tag with 256 B resolution, resulting in a larger degree of subblocking for configurations with smaller cache line size. The non-prefetching protocols (*32s*, *64s*, *128s*, *256*) have cache line sizes of 32, 64, 128 and 256 B respectively. The subblocked bundled capacity prefetching protocols (*32sbc*, *64sbc*, *128sbc*) have cache line sizes of 32, 64 and 128 B. All results are normalized relative to the *256* B configuration. The miss ratio for each application is indicated for the *256* B configuration.

ing, it would be easier to prefetch the lines that are available in the subblocks sharing the same address tag, rather than sequentially prefetching the lines with consecutive addresses. As a consequence, the subblocked version of the bundled capacity prefetching scheme becomes less complex to implement since the exact data to prefetch do not have to be specified. The prefetched lines are aligned with the addresses that have the same cache tag. On a non-communication miss, the bus request demands the owner of the cache line for all available lines that have the same cache tag.

6.1. Evaluation of the Subblocked Prefetching Schemes

The behavior of the subblocked prefetching schemes can be studied in Figure 3. To make a fair comparison, all configurations have a similar memory overhead cost. The cache tag size has a resolution of 256 B size in all configurations. A total of seven configurations has been tested. The configurations *32s*, *64s* and *128s* are ordinary non-prefetching subblocked schemes having 32, 64 and 128 B cache lines respectively. The *32s* configuration has 8 subblocks per tag, the *64s* configuration 4 subblocks per tag and the *128s* configuration 2 subblocks per tag. The protocol with a 256 B cache line *256* is not subblocked at all. The remaining three configurations *32sbc*, *64sbc* and *128sbc* are also subblocked to varying degrees and use the subblocked version of bundled capacity prefetching.

Since subblocking results in a larger number of cache misses due to an increase in capacity misses, the configurations with a small cache line size are more heavily subblocked than the ones with longer cache lines. For example, cache misses and traffic will increase more for the *32s* and *32sbc* configurations than for the *64s* configuration due to subblocking. A possibility would be to subblock less, but that would lead to an increased cache implementation cost. The subblocking results in less beneficial results for shorter cache line sizes and is therefore the main reason why a subblocked capacity prefetching configuration with a *very* short cache line is not suggestible.

On the average it seems like the lowest number of cache misses occurs when the cache line size is 64 B and subblocked bundled capacity prefetching is used. Compared with the non-prefetching 256 B configuration, the address traffic is somewhat increased (10 percent), while data traffic is lowered by more than 50 percent. The reason for the good results with the 64 B cache line size could be that all the benchmarks were optimized at compile time to run on machines with this cache line size.

Applications with a large amount of capacity misses experience a larger increase in traffic in subblocked bundled capacity prefetching than applications with less capacity misses. This is almost entirely caused by a larger number of write-backs, which generate more address and data traffic. The effect is more pronounced in subblocked caches.

7. Conclusion

This paper presents a method to simply categorize cache lines in communicating or non-communicating. The optimization proposed is to issue prefetch requests for non-communicating lines. No optimization has been carried out for communicating lines.

We have demonstrated how the strategies of capacity prefetching and bundling can be used to improve the effectiveness of static sequential prefetch schemes, while vastly reducing the overhead in address and data communication. The strategies reduce the capacity and cold misses by sequentially prefetching lines without causing problems with false sharing. The strategies create virtually longer cache lines for all non-communicating data and keep the cache lines short for communicating data. The hardware implementation cost of the prefetching scheme is very low. By combining the messages generated at each prefetch for all *Read* and *Upgrade* requests, the address traffic can be significantly lowered using bundled capacity prefetching.

This should not be viewed, however, as the ultimate result of these two methods, but rather an indication that they should be studied in the context of many more coherence schemes.

8. Future Work

Several authors have studied how to adaptively fetch data into a cache. Dahlgren et al. studied how to adaptively determine the prefetch distance in a multiprocessor [8]. An interesting study would be to combine an adaptive protocol with a capacity prefetch protocol.

It also would be interesting to use a separate dynamic prefetching strategy for communication data. That way, different effective cache line sizes would be applied to communication and capacity misses according to what seems to fit the running application best.

9. Acknowledgement

We would like to thank Jim Nilsson for providing us with the original version of the cache coherence protocol model for Simics and Martin Karlsson who helped us set up the commercial benchmarks. We also would like to express gratitude to the Department of Scientific Computing, Uppsala University, for letting us borrow their SunFire 15K server for simulation.

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

References

[1] http://ecperf.theserverside.com/ecperf/.

[2] http://www.spec.org/osg/jappserver2001/.

[3] http://www.spec.org/osg/jbb2000/.

[4] A. Agarwal, J. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems (TOCS)*, 6(4):393–431, 1988.

[5] 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.

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

[7] T.-F. Chen and J.-L. Baer. A Performance Study of Software and Hardware Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, 1994.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] M. Garzaran, J. Briz, P. Ibanez, and V. Vinals. Hardware Prefetching in Bus-Based Multiprocessors: Pattern Characterization and Cost-Effective Hardware. In *Proceedings of Parallel and Distributed Processing 2001*, pages 345–354, 2001.

[14] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffi c. In *25 Years of the International Symposia on Computer Architecture (selected papers)*, pages 255–262, 1998.

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] S. Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 160–169, 1990.

[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] 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.