Adaptive Coherence Batching for Trap-Based Memory Architectures

Håkan Zeffer and Erik Hagersten
Department of Information Technology, Uppsala University
P.O. Box 337, SE-751 05, Uppsala, Sweden
{hakan.zeffer, erik.hagersten}@it.uu.se

ABSTRACT

Both software-initiated and hardware-initiated prefetching have been used to accelerate shared-memory server performance. While software-initiated prefetching require instruction set and compiler support, hardware prefetching often require additional hardware structures or extra memory state.

The *coherence batching* scheme proposed in this paper keeps the system completely binary transparent and does not rely on any additional hardware. Hence, it can be implemented without additional hardware in software coherent systems and improve performance for already optimized and compiled binaries.

We have evaluated our proposals on a *trap-based memory architecture* where fine-grained coherence permission checks are done in hardware but the coherence protocol is run in software on the requesting processor. Functional full-system simulation shows that our software-only coherence-batch scheme is able to reduce the number of coherence misses with up to 60 percent compared to a system without coherence batching. The average miss reduction is 37 percent while the average bandwidth usage is reduced.

1. INTRODUCTION

Flexibility and dynamic coherence protocols have been used to enhance both software [8, 13, 35, 12, 33] and hardware [10, 29, 27, 20, 17, 19] shared-memory performance. It has been shown that per-application tailor-made coherence strategies [13], different coherence protocols [8, 33], coherence granularity and memory consistency model adjustments [35, 33] can be used to improve software distributed shared-memory (DSM) performance. Also many elaborate prefetch schemes have been proposed over the years [27, 17, 20, 19]. These typically try to reduce the coherence miss latency by fetching data and permission in advance. However, due to implementation complexity and potential slow-down for ill-suited applications they have rarely been implemented in hardware-based systems.

This paper explores different solutions for dynamically finding appropriate coherence unit sizes and identifying stable sharing patterns. The basic idea is to use this information to batch together coherence activities, and hence, reduce the number of coherence misses and potentially decrease the consumed bandwidth. We call this adaptive coherence optimization *coherence batching*. Our coherence-batching proposal is specifically targeting *trap-based memory architectures* (TMA), where the coherence permission check is done in hardware and the coherence protocol is run in software on the requesting processor [34]. All coherence-batch schemes presented in this paper are software-only, but like hardware prefetching, completely binary transparent to the system.

Software coherent systems are often more flexible than their hardware-only counterparts and should more easily be able to cope with extra complexity. However, there are a couple of potential problems with software coherent systems. For example, software coherence is only active for coherence misses and can thus only monitor activities that are associated with misses. Moreover, implementing associative structures in software are considerably more expensive than in hardware. Functional full-system multi-processor simulation show that our software-only coherence batching is able to reduce the number of coherence misses with up to 60 percent (37 percent on average) even though only misses can be observed by the predictor. Moreover, because of high batch accuracy and reduced coherence traffic, we find our scheme capable of reducing the consumed bandwidth with up to 22 percent (12 percent avg.).

The next section gives an overview of trap-based memory architectures and the basic coherence batch idea. Section 3 presents our simulation environment and the benchmarks used. Section 4 contains all our experiments and all our findings. Finally, we present related work and conclude in Sections 5 and 6.

2. COHERENCE BATCHING FOR TMA

We have earlier shown that different coherence protocols and coherence unit sizes can be used by software-coherent shared memory to outperform hardwired coherence [33]. It has also been shown that coherence unit size scaling can improve performance of a trap-based memory architecture with on average 20 percent [34]. A trap-based memory architecture detects fine-grained coherence violations in hardware, triggers

a coherence trap when one occurs and maintains coherence by software in coherence trap handlers. It has been argued that detection and handling of coherence violations can be implemented with a minimal amount of extra hardware support by exploiting the trap mechanisms already existing in modern microprocessors. In addition, the TMA proposal relies on the virtual memory system to track which memory pages are shared between multiple coherence domains [34]. The goal of this paper is to dynamically and transparently adjust coherence unit size, and hence improve system performance without any additional hardware.

The essence of coherence batching is to find the appropriate coherence unit size, and then use that information to reduce the number of coherence misses in the system. Our software coherence protocol and an appropriate home-node placement make it possible to look up multiple directory entries at the same time. However, data placement is directly dependent on the sharing behavior of an application. While we believe that data of stable sharing patterns will be placed together, we add additional control data to avoid unfair bandwidth reduction numbers.

3. PERFORMANCE EVALUATION DATA

This section describes our simulation methodology, simulator framework and our benchmarks.

3.1 Simulation Methodology

We use the Simics full-system simulator [24] simulating a SPARC-V9 system running a unmodified Solaris 9 operating system to collect multi-processor memory traces. These traces are feed into a custom, functional, inter-processor, directory-based coherence simulator. All coherence-batch optimizations evaluated in this paper are implemented inside this memory simulator. The simulator is used to collect statistics for the number of coherence misses, data- and control-packets sent. It is not capable of modeling timing nor the dynamic bandwidth.

3.2 Benchmarks

APACHE: Static Web Content Serving: We use Apache 2.0.43 configured to use a hybrid multi-process multi-threaded server model with 64 POSIX threads per server process [4]. Our experiments use a hierarchical directory structure of 80,000 files (with a total data size of approximately 1.8 GB) and a modified version of the Scalable URL Reference Generator (SURGE [5]) to simulate 6400 users (400 per processor) with an average think time of 12ms.

SPECjbb2000: SPECjbb2000 is a server-side Java benchmark that models a 3-tier system, focusing on the middleware server business logic and object manipulation [28]. The benchmark includes driver threads to generate transactions as well as an object tree working as a back-end. Our experiment use 24-driver threads (1.5 per processor) and 24-warehouses (with a total data size of approximately 500MB).

SPLASH-2: We have also chosen to study a subset of the well-known workloads from the SPLASH-2 benchmark suite [31]. The selected programs was chosen to represent a variety of communication and synchronization requirements.

Program	Problem Size
`apache`	1000 transactions
`fft`	256k points
`jbb`	5000 transactions
`lu-c`	512×512 matrices, 16×16 blocks
`lu-nc`	512×512 matrices, 16×16 blocks
`radix`	4M integers, radix 1024
`water-nsq`	512 molecules, 3 time steps
`water-sp`	512 molecules, 3 time steps

Table 1: Working set sizes of our benchmarks.

For example, we use `fft` because of its communication-intensive behavior and `radix` because its store access pattern is randomized.

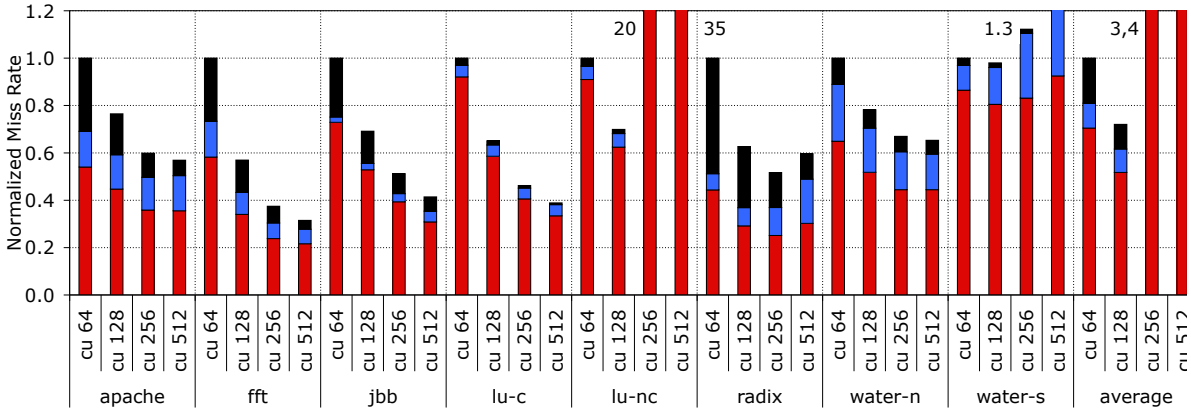
Data set sizes for the applications studied can be found in Table 1. All SPLASH-2 applications are compiled with gcc-3.4.3 (optimization level 3). PARMACS macros for locks and barriers are based on user-level *test&test&set* spin locks. Pause/Event macros are implemented with the POSIX Pthread library (only `radix` uses a small amount of pauses).

4. EXPERIMENTS AND RESULTS

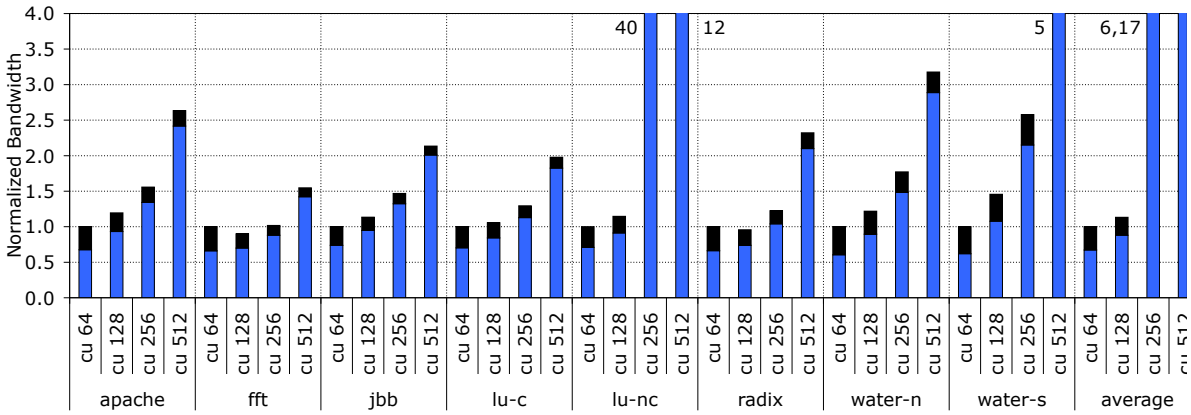
We have in earlier studies shown that the coherence unit size is very important for performance [33, 34]. However, it is well known that not all applications scale with a large coherence unit. In general, the miss rate and the traffic as a function of coherence unit size follows a U-shaped curve. A large size reduces the number of cold misses but often increases the amount of false sharing [11]. We use coherence unit size scaling as a comparison base for our evaluated optimizations. Our goal is a coherence-batching scheme that reduces the coherence misses more than the optimal coherence unit size and that consumes less bandwidth than the best coherence unit size (in terms of low bandwidth consumption).

4.1 Coherence Unit Scaling

Figure 1 (a) shows normalized miss rate as a function of coherence unit size (varied between 64 and 512 bytes). We have broken down the miss rate into three different parts corresponding to read misses (*rd*), write misses (*wr*) and upgrade misses (*upgr*). We find the miss rate for most of the applications to scale well with coherence unit size. However, `lu-nc`, `radix` and `water-s` are penalized by false sharing, and hence, their corresponding miss rates are increased for large coherence unit sizes. Figure 1 (b) shows the consumed data- (*data*) and control- (*ctrl*) bandwidth while the coherence unit size is scaled. Some benchmarks (`fft` and `radix`) actually decrease their bandwidth consumption with a coherence unit size of 128 bytes. Almost no unnecessary data is fetched and control bandwidth is decreased because of less directory accesses for these benchmarks. However, all the tested applications consume more than 50 percent of extra bandwidth when a coherence unit size of 512 bytes is used. The applications that suffer from false sharing show a very high degree of bandwidth consumption. To summarize, Figure 1 (a) and (b) show that a large coherence unit size can enhance system performance for applications that



(a) Normalized miss rate.



(b) Normalized bandwidth consumption.

Figure 1: Coherence unit size scaling (CU). All results are normalized to the base system with a coherence unit size of 64 bytes. (16 processor runs.)

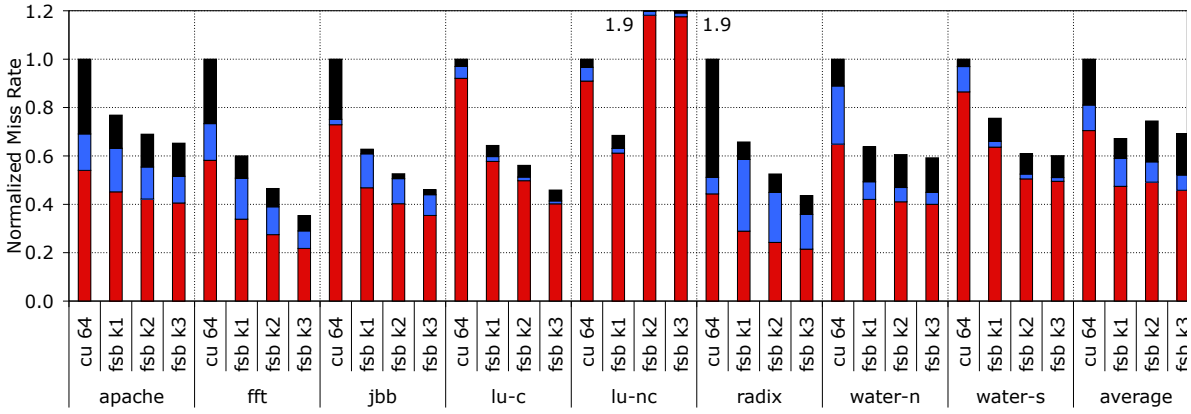
can handle a high bandwidth consumption and that do not suffer from false sharing.

4.2 Fixed-Sequential Coherence Batching

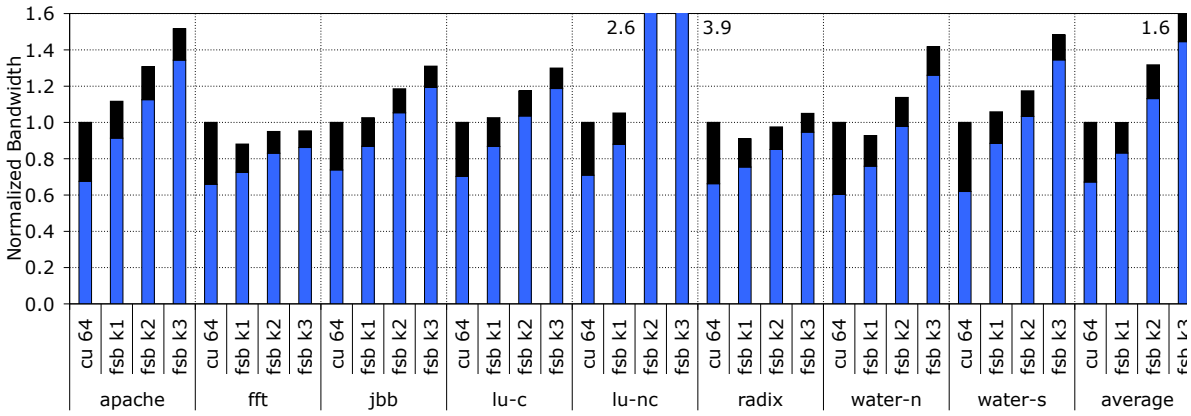
Sequential prefetching has been proposed to enhance multiprocessor performance [11]. However, the fixed-sequential prefetching scheme, as proposed by Dahlgren et al, requires that the memory system is able to handle three new network commands: a prefetch request and two reply messages [11]. We have implemented a *fixed-sequential coherence-batching* (FSB) scheme based on Dahlgren et als' proposal. FSB can be implemented entirely in software, and hence, it eliminates the need for additional hardware. The basic idea is to get permission for $k + 1$ consecutive coherence units with a single batched directory update when a coherence miss is encountered.

Even though the FSB scheme is very simple, it is very effective for applications where the cold and true-sharing misses exhibit a high degree of spatial locality. These misses are expected to be reduced as much as if a coherence unit size two times larger than the original one where used [11]. Note that this is only true if an application has high spatial locality and the direction of the spatial memory streams move towards higher addresses. Moreover, the false-sharing miss rate component is expected to be the same as with the original coherence unit size [11]. This holds if the false-sharing behavior is not too fine grained and not ill aligned.

Figure 2 (a) shows the normalized miss rate for three different degrees of batching. We find it very interesting that the miss rate for the applications that suffer from false-sharing with a large coherence unit size, `lu-nc`, `radix` and `water-n`, performs much better when our fixed-sequential coherence-batching scheme is used. However, FSB can not fully remove the false-sharing misses in the `lu-nc` application. Figure 2 (b) shows that also the bandwidth consumption is kept low when compared to a large coherence unit size. Note that



(a) Normalized miss rate.



(b) Normalized bandwidth consumption.

Figure 2: Fixed-sequential coherence batching (FSB) with the batch degree k varied between one and three. All results are normalized to the base system (CU) with a coherence unit size of 64 bytes. (16 processor runs.)

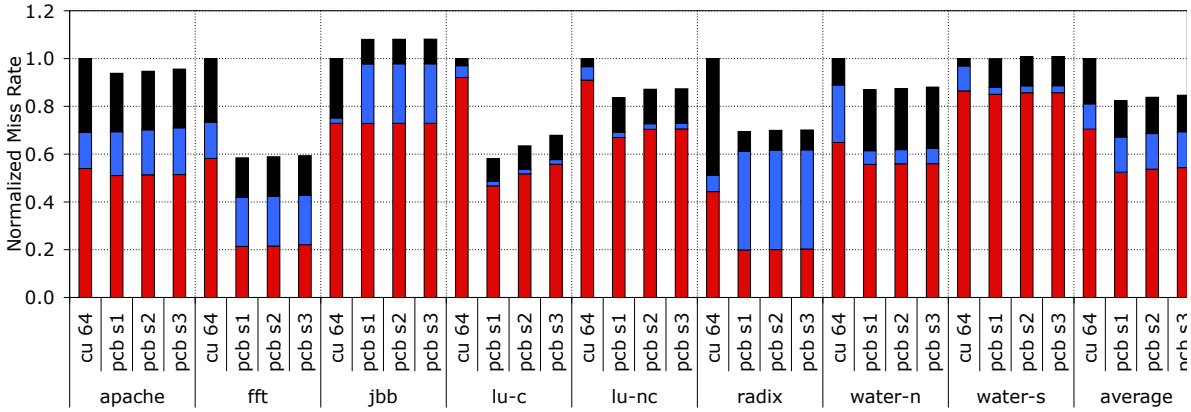
fixed-sequential coherence-batching with a batch degree of three ($k = 3$) is to be compared to a coherence unit size of 256 bytes since the same amount of data is fetched per miss.

While this scheme works very well for most applications, it is not able to hide all false sharing, nor keep the consumed bandwidth under control when a batch degree larger than one is used. We conclude that this technique works better than coherence unit size scaling. However, it is not stable when exposed for extensive false sharing.

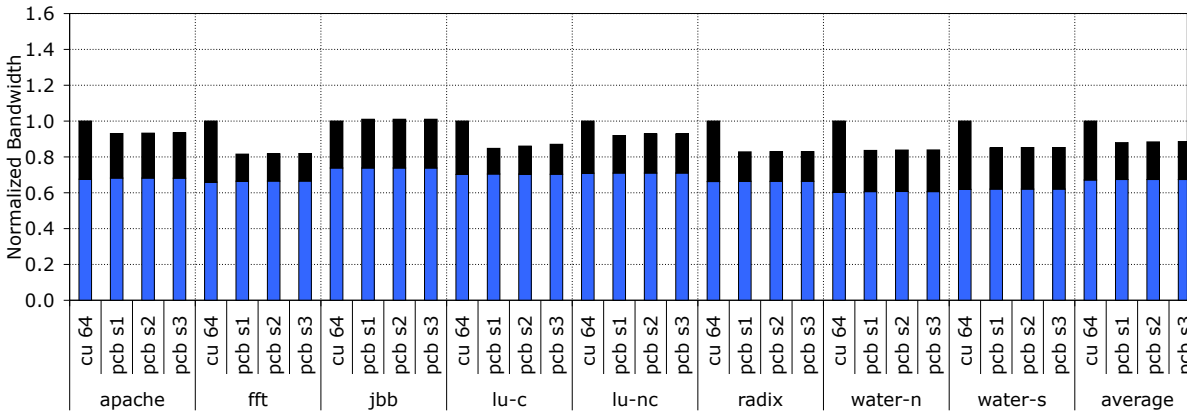
4.3 PC-Based Coherence Batching

It has been shown that very few loads and stores are responsible for most of the cache misses [1]. This observation has led to multiple instruction-based prediction schemes; for example, the one presented by Chen and Baer [9] and the one presented by Kaxiras and Goodman [17]. We find that more than 10 percent of all coherence misses in `apache`, `fft` and `water-s` correspond to a single program counter (PC). Even more encouraging, this number is even higher for `lu-c`, `lu-nc` and `radix` where more than 30 percent of all coherence misses correspond to a single PC.

To our knowledge, all earlier proposed instruction-based schemes rely on plentiful of dedicated hardware. In particular, those techniques require that both the PC of the instruction that generates a coherence event and information from the cache coherence mechanism are available to the hardware predictor [17]. Our *PC-based coherence-batching* (PCB) implementation does not rely on any additional hardware. All data are processor private and located in small main memory data structures. Moreover, while Kaxiras and Goodman use three events as input to their predictors: (1) a cache miss, (2) write fault, and (3) external cache read/invalidation, we only look at coherence misses. We plan to run all our batching code inside TMA's



(a) Normalized miss rate.



(b) Normalized bandwidth consumption.

Figure 3: PC-based coherence batching (PCB) with the stability threshold *s* varied between one and three. All results are normalized to the base system (CU) with a coherence unit size of 64 bytes. (16 processor runs.)

coherence miss handlers. Note that the PC of the trapping instruction can be found as the "return address" on the stack for implementations similar to our earlier TMA proposal [34].

The idea behind our PC-based coherence-batching scheme is to keep track of the misses that correspond to a single PC. If the pattern of these misses are stable, the PC is classified as a coherence-batch PC, and hence, at the next miss, coherence batching of the observed pattern is used. We log information about misses corresponding to a single PC and a 512 byte memory region in a small PC-indexed hash structure. When a coherence miss (by the corresponding PC) to a new memory region occurs, we simply change the log focus to the new 512 byte area. However, at each region switch, we check if the recorded miss pattern of the previous memory region can be classified as stable. We use the pattern history to check for miss pattern stability. Only if the same miss pattern occurs a predefined number of times (the "stability threshold") we classify it as stable, and hence, classifies its corresponding PC as a coherence-batch PC.

We have tested different pattern stability thresholds (in this study: 1, 2 and 3) before we upgrade a PC to coherence-batch status. To dynamically adapt to sharing changes, we automatically downgrade a PC after eight miss-triggered coherence-batch events. However, since the PC's miss pattern has been classified as stable before, we only require a single pattern-stability test to move the PC back to coherence-batch status.

The alignment of the memory region under log focus is important for the stability test. Letting the first coherence miss decide the region alignment improves the number of stable patterns found in the majority of the tested applications. For example, the stable pattern for `fft`'s most important

PC, responsible for 17 percent of its coherence misses, occurs in 85 instead of 64 percent of all area switches. Our PCB scheme targets loop-counter indirect coherence misses, and hence, we have found that alignment for positive coherence-stream patterns are enough for most applications.

Figure 3 (a) and (b) show the miss rate and the bandwidth consumption, for PCB while varying the stability threshold *s*. Since this optimization targets loops, it works much better for the scientific codes than for the static web content service and the server-side java benchmark. We find it very encouraging that the bandwidth consumption is kept low. While our PC-based coherence-batch technique reduces the average miss rate with 18 percent it also reduces the average consumed bandwidth with 12 percent, when compared to the base system and a coherence unit size of 64 bytes. The bandwidth reduction comes from a high batch accuracy in combination with less coherence traffic.

4.4 Neighborhood-Based Coherence Batching

Neighborhood prefetching is also a instruction-based prediction scheme [19]. A group of coherence units surrounding the missing one, its *neighborhood*, is prefetched in parallel with the demand-fetch. The implementation proposed by Koppelman collects meta data in hardware and relies on both the coherence input as well as the PC, that caused the coherence miss [19]. Our *neighborhood-based coherence-batching* (NB) proposal can be implemented completely in software in a TMA system's coherence trap handlers. While our implementation is based on the faulting PC and memory areas, just as PC-based coherence batching, we use a different schema in order to collect miss-pattern information. Instead of logging coherence misses on the fly, as in all previous instruction-based proposals, we check the directory state for the old memory region at region switches. This makes it possible to avoid the potential overhead introduced by logging and to avoid migratory upgrade misses etc. However, an extra directory lookup might be needed while performing the stability check for a PC.

To dynamically adjust the batching strategy, we use a stability threshold for upgrades and downgrade a PC after eight batching events. Just as in PCB, we use a single stability check before we re-upgrade a PC to batch mode. Note that the PC under stability test can be selected with sampling, since we are targeting PCs that are responsible for a large amount of coherence misses.

Figure 4 (a) shows the miss rate for NB while varying the stability threshold *s* between one and three. The most aggressive configuration that only require a single stability check before a PC is classified for coherence batching shows a very impressive reduction in the number of coherence misses. For example, the miss rate of `apache` (`jbb`) is decreased with almost 40 (60) percent when compared to the base system and a coherence unit size of 64 bytes. However, Figure 4 (b) shows that the bandwidth consumption is not stable with the threshold set to one. Since the average miss rate reduction for the NB system with a stability threshold of two is almost 40 percent (avg.) and the bandwidth usage is kept below the baseline system (avg.), we find this system configuration to be a better choice.

4.5 Meta-Data Size and Layout

The size and the design of PCB's and NB's corresponding meta data can of course affect system performance. For example, implementing associative structures in software are considerably more expensive than in hardware. Moreover, big data structures in main memory may decrease cache performance. Hence, it is important to keep both the size and the associativity of meta-data small.

Figure 3 and 4 report PCB and NB numbers based on 1024 meta-data entries and fully associative structures. Figure 5, on the other hand, shows the miss rate for NB while varying the number of meta-data entries per processor in the system. The performance of the scientific applications is very high even if few entries are used. However, `apache` and `jbb` clearly need more entries than the scientific applications to perform. We find a 32 entry implementation capable of reducing the average miss rate with 30 percent.

The results in Figure 5 corresponds to fully associative meta-data structures. Figure 6 shows the normalized miss rate when associativity is varied between direct-mapped and fully-associative, when a 32 entry configuration is used. We find it very encouraging that a direct mapped or a two-way set associative structure can be used without any performance degradation at all.

4.6 Result Summary

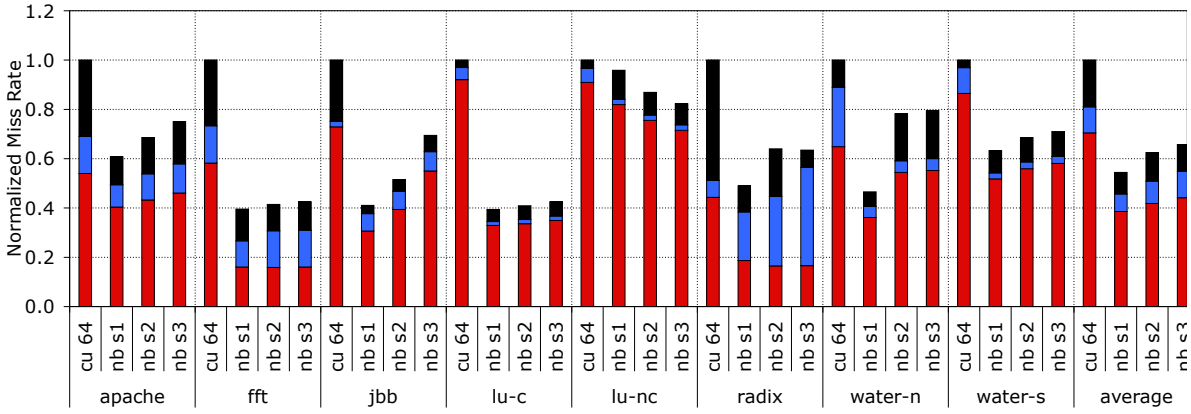
Figure 7 (a) and (b) compare the miss rate and the consumed bandwidth for our coherence-batching proposals to the baseline system with a coherence unit size of 64 as well as 128 bytes. The `cu 64` and the `cu 128` bar corresponds to the baseline system with 64 and 128 bytes as coherence unit size. Moreover, we picked the best configuration for each coherence-batch scheme.

FSB (`fsb k1`) shows both low miss rate numbers and consumed bandwidth when compared to the base system with a coherence unit size of 64 or 128 bytes. This is mainly because of less false-sharing than in the corresponding coherence unit scaled system. However, we know that FSB is not stable when exposed for extensive false sharing. Our selected PC-based coherence-batch configuration (`pcb s1`) reduces the miss rate with 18 percent while keeping the consumed bandwidth low. If bandwidth, is a big issue and scientific codes is to be run, this is the system configuration of choice. Finally, neighborhood-based coherence-batching (`nb s2`) shows the lowest miss rate numbers, while reducing the bandwidth when compared to the base-line system.

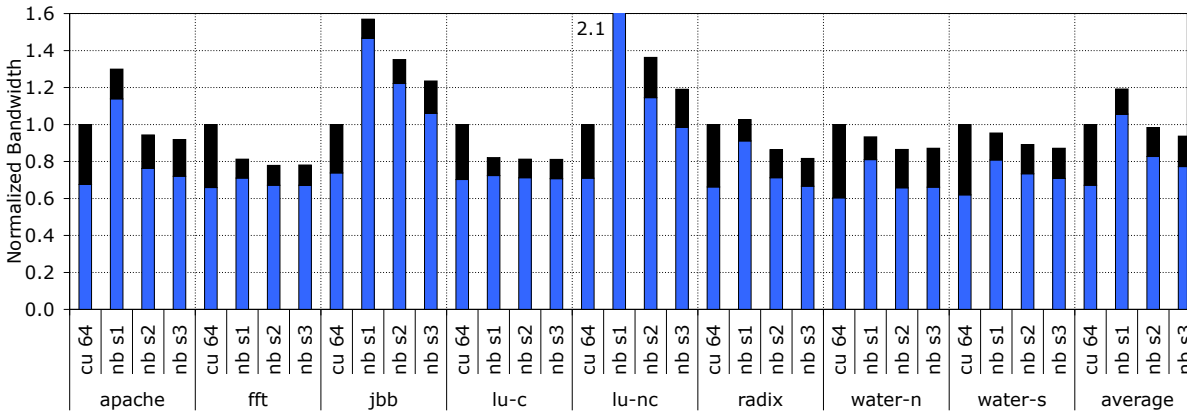
Note that the miss rate for both FSB and PCB are higher than the corresponding `cu 128` configuration for both `fft` and `radix`. The reason for this is that `fft` contains some memory streams that move towards lower addresses and because `radix` has a store pattern that is highly randomized. Hence, a large coherence unit size performs better than our batching schemes that are optimized for streams toward higher addresses.

5. RELATED WORK

It has been found that there exists many kinds of cache invalidation patterns in shared memory programs [14]. Hence,



(a) Normalized miss rate.



(b) Normalized bandwidth consumption.

Figure 4: Neighborhood-based coherence batching (NB) with the stability threshold s varied between one and three. All results are normalized to the base system (CU) with a coherence unit size of 64 bytes. (16 processor runs.)

many adaptive coherence protocols that target these coherence patterns have been proposed over the years [29, 10, 15]. Address-based prediction schemes [27, 20] emerged from two level adaptive branch prediction as proposed by Yeh and Patt [32]. Mukherjee and Hill showed that address-based prediction in coherence protocols can be generalized by using two-level adaptive predictors located at the directory [27]. The predictor remember general coherence patterns and predict what messages to send in advance. Lai and Falsafi improve Mukherjee's and Hill's work with the "memory sharing predictor" [20].

Abraham et al show that very few loads and stores are responsible for most cache misses [1]. This observation has led to multiple instruction-based prediction schemes; including cache management [30] and memory dependence predictions [26]. Chen and Baer [9] were the first to propose instruction-based prediction for parallel systems. Kaxiras et al propose instruction-based prediction as a general technique to optimize hardware shared-memory systems while tracking instruction history in relation to coherence events [17, 18]. Neighborhood prefetching is a similar technique where offsets of accesses that subsequently miss are recorded and later on used for prefetching at later accesses by the same static instruction [19].

Different kinds of prediction have also been used for dynamic self invalidation [21] (first proposed by Lebeck and Wood [23]), dead-block prefetching [22], owner [2] and destination-set prediction [25].

Prefetching have also been used in software-coherent systems. Multiple software distributed shared-memory systems implement address-based prefetching schemes that rely on past history of memory access faults. For example, Karlsson and Stenström use a history-based heuristic together

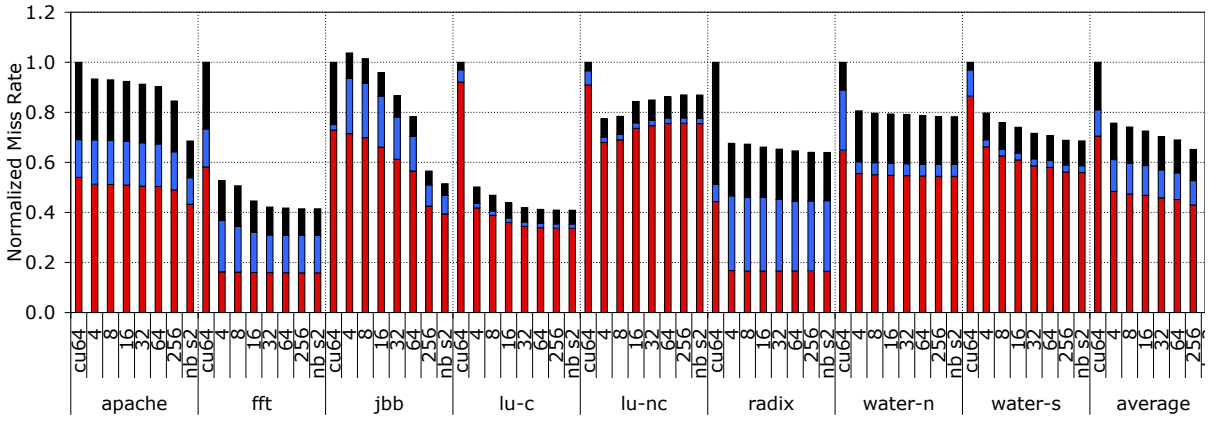


Figure 5: Normalized miss rate with the number of meta-data PC entries varied from four to 256. NB performance with the stability threshold set to two. All results are normalized to the base system (CU) with a coherence unit size of 64 bytes. (16 processor runs.)

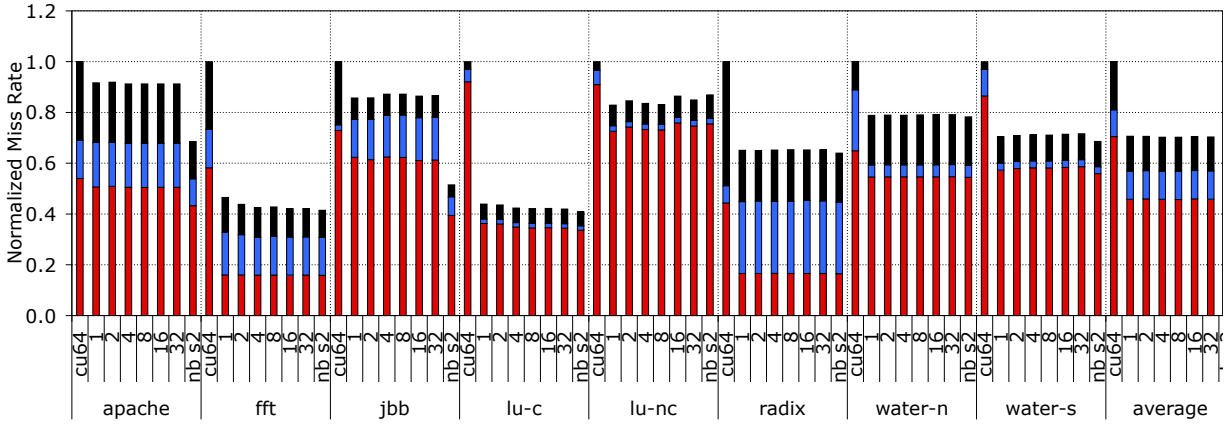


Figure 6: Normalized miss rate with the associativity of the meta-data varied from one to 32. NB performance with the stability threshold set to two and 32 meta-data entries. All results are normalized to the base system (CU) with a coherence unit size of 64 bytes. (16 processor runs.)

with sequential prefetching in a multiple-writer page-based shared-memory system [16]. Bianchini et al use the past history of memory access faults do adapt between *repeated-phase* and *repeated-stride* prefetching [6, 7]. To enhance shared virtual memory performance, Amza et al use *dynamic aggregation*. That is, by aggregating pages that are sequential in virtual memory into page groups, data is implicitly prefetched [3].

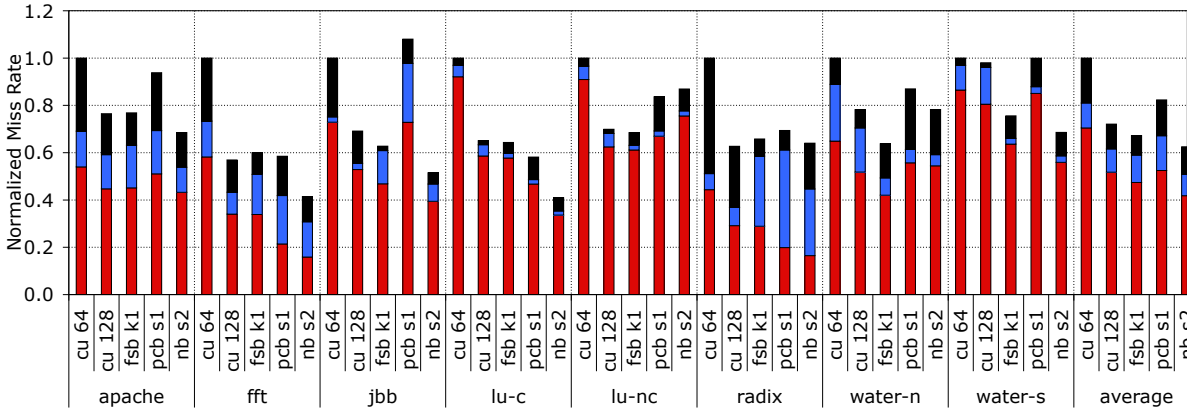
The coherence batching described in this paper differs from proposed hardware based prefetching in multiple ways. For example, our technique only takes coherence misses into account, batches together directory updates and data fetches instead of issuing multiple prefetching requests. Moreover, our batching scheme target trap-based memory architectures and does not require any additional hardware support at all.

Also when compared to software-coherent systems does our proposal differ. For example, coherence batching introduce instruction-based prediction in software-coherent systems. Moreover, all prediction schemes shown so far in software distributed shared memory have targeted shared virtual memory systems. The system we are targeting with our coherence batching maintain coherence at a much finer granularity.

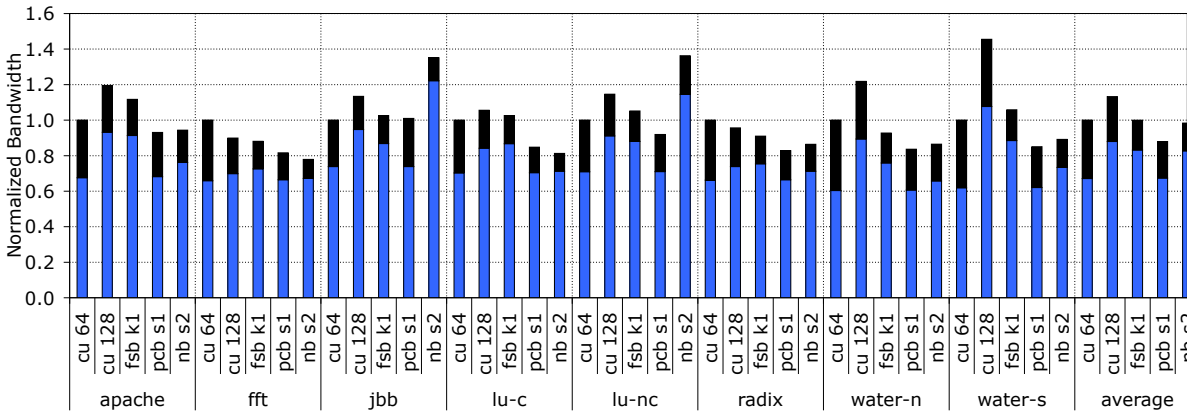
6. CONCLUSIONS

Many elaborate prefetching schemes have been proposed for shared-memory systems. The idea behind these schemes is to reduce coherence miss latency by fetching data and permission in advance. However, due to implementation complexity and potential slow-down for ill-suited applications they have rarely been implemented in hardware-based systems.

This paper explores different solutions for dynamically finding appropriate coherence unit sizes and stable sharing patterns. The basic idea is to use this information to batch together coherence activities, and hence, reduce the number of coherence misses and potentially decrease the consumed bandwidth. Our *coherence batch* strategy differ from previously proposed schemes because it can be implemented completely in software, but still, transparently



(a) Normalized miss rate.



(b) Normalized bandwidth consumption.

Figure 7: Coherence unit size scaling (CU), fixed-sequential coherence batching (FSB), PC-based coherence batching (PCB) and neighborhood-based coherence batching (NB) performance. All results are normalized to the base system with a coherence unit size of 64 bytes. (16 processor runs.)

run already compiled and optimized binaries. This paper evaluates multiple coherence batching proposals in the context of a *trap-based memory architecture*. Trap-based memory architectures maintain fine-grained coherence permission checks in hardware but run the coherence protocol in software coherence-trap handlers on the requesting processor.

It has earlier been shown that a small number of load and store instructions corresponds to the majority of cache and coherence misses. This paper shows that this finding can be used by software to reduce the number of coherence misses. We find our coherence-batch scheme targeting loops very efficient for scientific codes, especially in terms of bandwidth reduction. Our more aggressive directory-based neighborhood batching reduces the number of coherence misses even further while keeping the bandwidth consumption under control. This scheme also improves the performance of the commercial applications studied. Functional full-system simulation shows that our software-only coherence batching is able to reduce the number of coherence misses with up to 60 percent and the consumed bandwidth with up to 22 percent. The average numbers are 37 and 12 percent respectively. We find it very encouraging that our program-counter-based coherence-batch schemes are stable and do not get penalized by false sharing or fine grained coherence patterns.

7. ACKNOWLEDGMENTS

We would like to thank David Wood and Mark Hill at University of Wisconsin for providing us with the `apache` and the `jbb` benchmarks. This work is supported by Sun Microsystems, Inc., and Parallel and Scientific Computing Institute (PSCI), Sweden.

8. REFERENCES

[1] S. Abraham, R. Sugumar, D. Windheiser, B. Rau, and R. Gupta. Predictability of Load/Store Instruction Latencies. In *Proceedings of the 26th IEEE/ACM International Symposium on Microarchitecture (MICRO-26)*, pages 139–152, Dec. 1993.

[2] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in a cc-NUMA Architecture. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*, Nov. 2002.

[3] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '97)*, pages 90–99, June 1997.

[4] The Apache Software Foundation. *Apache HTTP Server Version 2.0.43 Reference Manual*.

[5] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the 1998 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 151–160, June 1998.

[6] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. D. Maria, M. Abud, and C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 198–209, Oct. 1996.

[7] R. Bianchini, R. Pinto, and C. L. Amorim. Data prefetching for software DSMs. In *Proceedings of the 12th International Conference on Supercomputing (ICS'98)*, pages 385–392, July 1998.

[8] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 152–164, Oct. 1991.

[9] 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 (ISCA'94)*, pages 223–232, Apr. 1994.

[10] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 98–108, May 1993.

[11] 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, July 1995.

[12] S. Dwarkadas, K. Gharachorloo, L. I. Kontothanassis, D. J. Scales, M. L. Scott, and R. Stets. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In *Proceedings of the 5th International Symposium on High-Performance Computer Architeture (HPCA-5)*, pages 260–269, Jan. 1999.

[13] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing (SC'94)*, pages 380–389, Nov. 1994.

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

[15] A. Kägi, N. Aboulenein, D. Burger, and J. R. Goodman. Techniques for Reducing Overheads of Shared-Memory Multiprocessing. In *Proceedings of the 9th International Conference on Supercomputing (ICS'95)*, pages 11–20, July 1995.

[16] M. Karlsson and P. Stenström. Effectiveness of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems. *Journal of Parallel and Distributed Computing*, 43(2):79–93, July 1997.

[17] S. Kaxiras and J. R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the 5th International Symposium on High-Performance Computer Architeture (HPCA-5)*, pages 161–170, Jan. 1999.

[18] S. Kaxiras and C. Young. Coherence Communication Prediction in Shared-Memory Multiprocessors. In *Proceedings of the 6th International Symposium on High-Performance Computer Architeture (HPCA-6)*, pages 156–167, Jan. 2000.

[19] D. M. Koppelman. Neighborhood Prefetching on Multiprocessors Using Instruction History. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 123–132, Oct. 2000.

[20] A.-C. Lai and B. Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, pages 139–148, May 1999.

[21] A.-C. Lai and B. Falsafi. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, pages 139–148, June 2000.

[22] A.-C. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*, pages 144–154, June 2001.

[23] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 48–59, June 1995.

[24] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[25] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*, pages 206–217, June 2003.

[26] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 181–193, June 1997.

[27] S. S. Mukherjee and M. D. Hill. Using Prediction To Accelerate Coherence Protocol. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*, pages 179–190, June 1998.

[28] Standard Performance Evaluation Corporation. *SPECjbb2000, A Java Business Benchmark*. White Paper.

[29] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 109–118, May 1993.

[30] G. Tyson, M. Farrens, and J. M. A. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th IEEE/ACM International Symposium on Microarchitecture (MICRO-28)*, pages 93–103, Nov. 1995.

[31] 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 (ISCA'95)*, pages 24–36, June 1995.

[32] T.-Y. Yeh and Y. N. Patt. Alternative Implementations of two-level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, pages 124–134, May 1992.

[33] H. Zeffer, Z. Radović, and E. Hagersten. Flexibility Implies Performance. Technical Report 2005-013, Department of Information Technology, Uppsala University, Apr. 2005.

[34] H. Zeffer, Z. Radović, M. Karlsson, and E. Hagersten. TMA: A Trap-Based Memory Architecture. Technical Report 2005-015, Department of Information Technology, Uppsala University, May 2005.

[35] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '97)*, pages 193–205, June 1997.