Aleksandar Milenkovic
University of Belgrade

In bus-based shared-memory multiprocessors, several techniques reduce cache misses and bus traffic, the key obstacles to high performance.

Achieving High Performance in Bus-Based Shared-Memory Multiprocessors

Bus-based shared-memory multiprocessors, or symmetric multiprocessors, are widely used in small- to medium-scale parallel machines of up to 30 processors. Their popularity has greatly increased because almost all modern microprocessors support cost-effective bus-based SMPs, making them the dominant architecture in parallel machines today.[1]

In bus-based SMPs, cache misses and bus traffic pose key obstacles to high performance. To overcome these problems, several techniques have been proposed. Cache prefetching, read snarfing, software-controlled updating, and cache injection reduce cache misses; migrate-on-dirty, adaptive migratory detection, load-exclusive instruction, and exclusive prefetching reduce invalidation bus traffic.

SMP performance

Private caches are essential for reducing bus congestion and for coping with memory reference latency in bus-based SMPs. Snooping cache coherence protocols can effectively keep shared data coherent because they use the simple and effective broadcast capability of a single bus.[2] Cache coherence protocols fall into two broad classes—write-invalidate and write-update —depending on whether a write to a shared cache block invalidates or updates all other copies of the block.[3] Although a write-update protocol produces fewer cache misses, the bus's higher write traffic often decreases overall performance.[4] Therefore, almost all commercial machines use write-invalidate as the standard protocol, preserving precious communication bandwidth.[5]

However, the widening gap between processor and memory speeds, the high contention on the bus, and data sharing in parallel programs cause two performance bottlenecks: large read and write cache-miss latencies and bus traffic. Although appropriate write buffers and relaxed memory consistency models can often hide

1092-3063/00/$10.00 © 2000 IEEE

Table 1. Techniques for reducing read misses and invalidation bus traffic.

Technique	Effects on read misses	Effects on bus traffic	Controlled by
Cache prefetching	Decrease: cold and replacement	Increase	Hardware or software
	Decrease, no change, or increase: coherence		
Read snarfing	Decrease: coherence	Decrease	Hardware
Software-controlled updating	Decrease: coherence	Decrease, no change, or increase	Software
Cache injection	Decrease: cold, coherence, and replacement	Decrease, no change, or increase	Software
Migrate-on-dirty	Increase or no change: coherence	Decrease, no change, or increase	Hardware
Adaptive migratory detection	No change	Decrease	Hardware
Load-exclusive instruction	No change	Decrease	Software
Exclusive prefetching	Increase or no change: coherence	Decrease	Software

Table 2. Hardware needed to reduce read misses and invalidation bus traffic.

Technique	Additional bits per cache line	Additional mechanisms per cache	Additional instructions
Software-controlled cache prefetching	None	Write-buffer or additional prefetch buffer for buffering of outstanding prefetch requests	Prefetch
Hardware-controlled cache prefetching	Several bits, depending on particular scheme	Write-buffer or additional prefetch buffer for buffering of outstanding prefetch requests	None
		and	
		hardware for address pattern detection and prefetch scheduling	
Read snarfing	None	Negligible modification of the snooping mechanism	None
Software-controlled updating	Private bit	Write-buffer or additional update buffer for buffering of outstanding update requests	Update and store-update
Cache injection	None	Injection table and negligible modification of the snooping mechanism	OpenWin, closeWin, update, and store-update
Migrate-on-dirty	None	Negligible modification of the bus control unit	None
Adaptive migratory detection	Two bits (three new states)	Modification of the snooping mechnism and migratory line on the bus	None
Load-exclusive instruction	None	None	Load-exc
Exclusive prefetching	None	Write-buffer or additional prefetch buffer for buffering of outstanding prefetch-exclusive requests	Prefetch-exc

write-miss latency,[6] read-miss latency still remains. Consequently, achieving high performance depends on reducing the number of read misses and bus traffic.

Several techniques based on snooping write-invalidate cache coherence protocols can reduce read misses and bus traffic in bus-based SMPs. As Table 1 shows, these techniques vary according to their effect on read misses, their effect on bus traffic, and whether software or hardware controls them. Table 2 shows additional hardware needed to support the techniques, compared to the basic SMP, which assumes use of lockup-free caches and the Illinois cache-coherence protocol.[2]

Cache prefetching, read snarfing, software-controlled updating, and cache injection all reduce read misses, which are classified as

- *cold misses*, which occur when the processor has never referenced the requested block before;
- *coherence misses*, which occur when the processor references the block but another processor has written to it; or

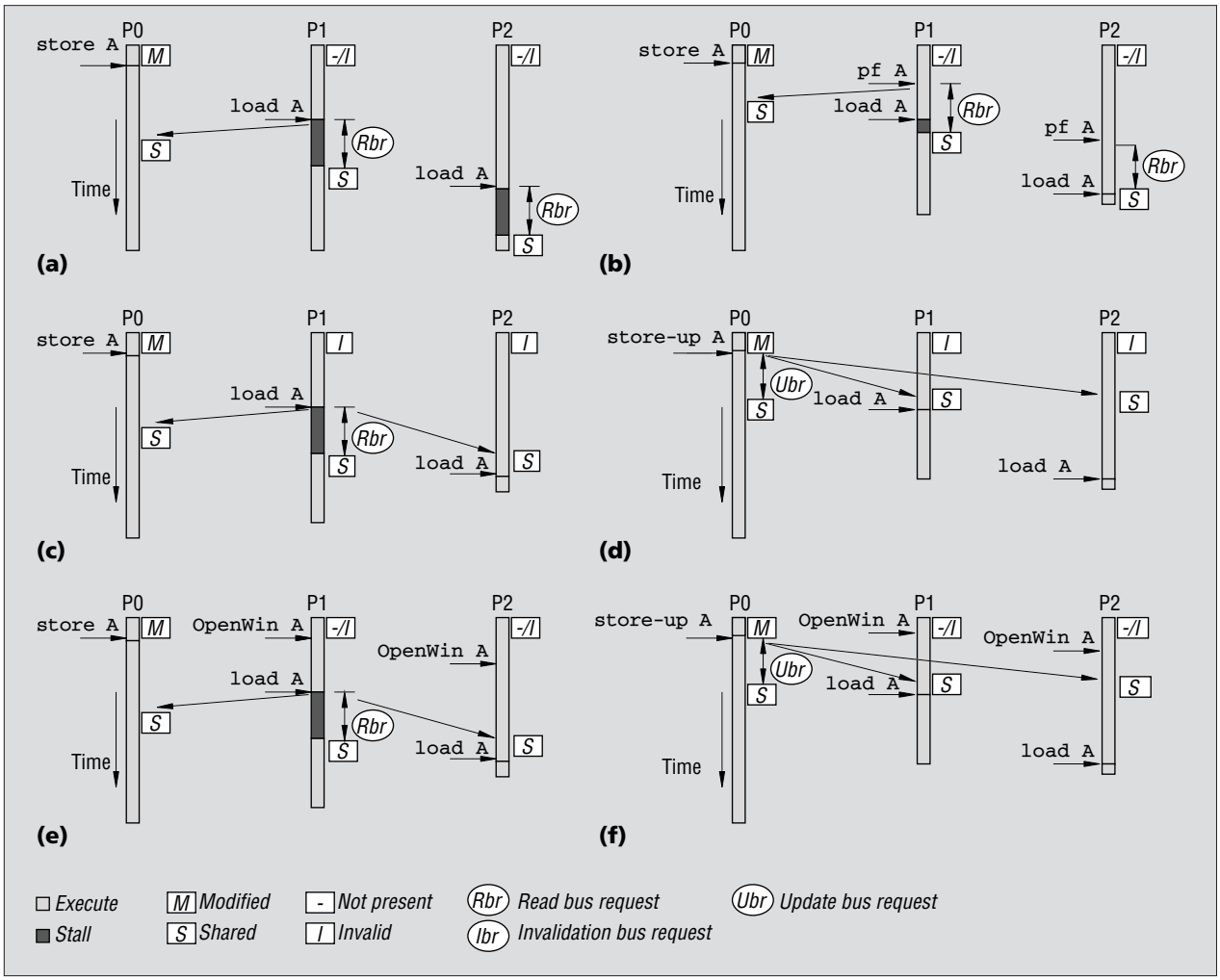

Figure 1. Read–miss reduction techniques: (a) base, (b) cache prefetching, (c) read snarfing, (d) software-controlled updating, (e) injection on first read, and (f) injection on update.

- *replacement misses*, or all other misses, which result from replacements due to the cache's limited size and associativity.

Figure 1 illustrates how each technique works in a simple producer-consumer sharing pattern: Processor P0 produces the data and processors P1 and P2 consume it.

Migrate-on-dirty, adaptive migratory detection, load-exclusive instruction, and exclusive prefetching reduce invalidation bus traffic, an important contributor to bus delays, especially in applications where migratory sharing prevails. In migratory sharing, many processors read and modify a block—but only one processor at a time. Unfortunately, if a processor reads a migratory block before it is written, the common write-invalidate protocol requires two separate bus transactions: a read request with block transfer and an invalidation request.

These techniques merge ownership acquisition with read-miss service, providing two benefits: They eliminate invalidation traffic under both release and sequential consistency, and they reduce write latency under sequential consistency. Figure 2 illustrates how each technique works in simple migratory sharing: Processor P0 initially modifies the block in its cache, and processor P1 reads and then writes the same block.

Cache prefetching

Cache prefetching is a common technique for hiding high memory latency in both multiprocessors and uniprocessors. It predicts which blocks, currently missing in the cache, will be referenced in the future and brings them into the cache prior to references triggering the misses. The returned block is not bound; it is still subject to invalidations by the cache coherence mechanism.

Cache prefetching can be either hardware- or software-controlled. Hardware-controlled cache prefetching is based on data access regularity in applications, assuming that data accesses in the near future will follow detected patterns. There are many hardware prefetching schemes—such as fixed and adaptive sequential prefetching and stride prefetching.[7] Although these techniques relieve the programmer or compiler of deciding what and when to prefetch, the hardware's knowledge of data access patterns limits their effectiveness. Additionally, except for the simplest sequential prefetching, all proposed schemes require complex hardware support, which is not yet found in microprocessors and multiprocessors.[1]

However, almost all modern microprocessors have instructions that support prefetching, so I concentrate on software-controlled cache prefetching. Here, a processor executes a special prefetch

instruction that moves a data block—that the processor is expected to use—into the cache before it is actually needed.[8] Prefetch instructions are nonblocking: The processor proceeds with program execution during fetching of the prefetched data. In the best case, the data block arrives at the cache before the processor needs it, and the load instruction results in a hit, as shown in processor P2 of Figure 1b. Cache prefetching can be useful even if the prefetch is not issued early enough, in which case the memory latency is only partially hidden, as shown in processor P1 of Figure 1b.

Software-controlled prefetching requires both hardware and compiler support. Hardware support includes prefetch instructions and a cache, which allows buffering of the prefetch requests in the write buffer or in an additional prefetch buffer. For compiler support, several compiler algorithms have been proposed to support software-controlled prefetching for applications based on numerical arrays[8] and on recursive data structures.[9]

Dean Tullsen and Susan Eggers examined the potential for software-controlled cache prefetching in bus-based

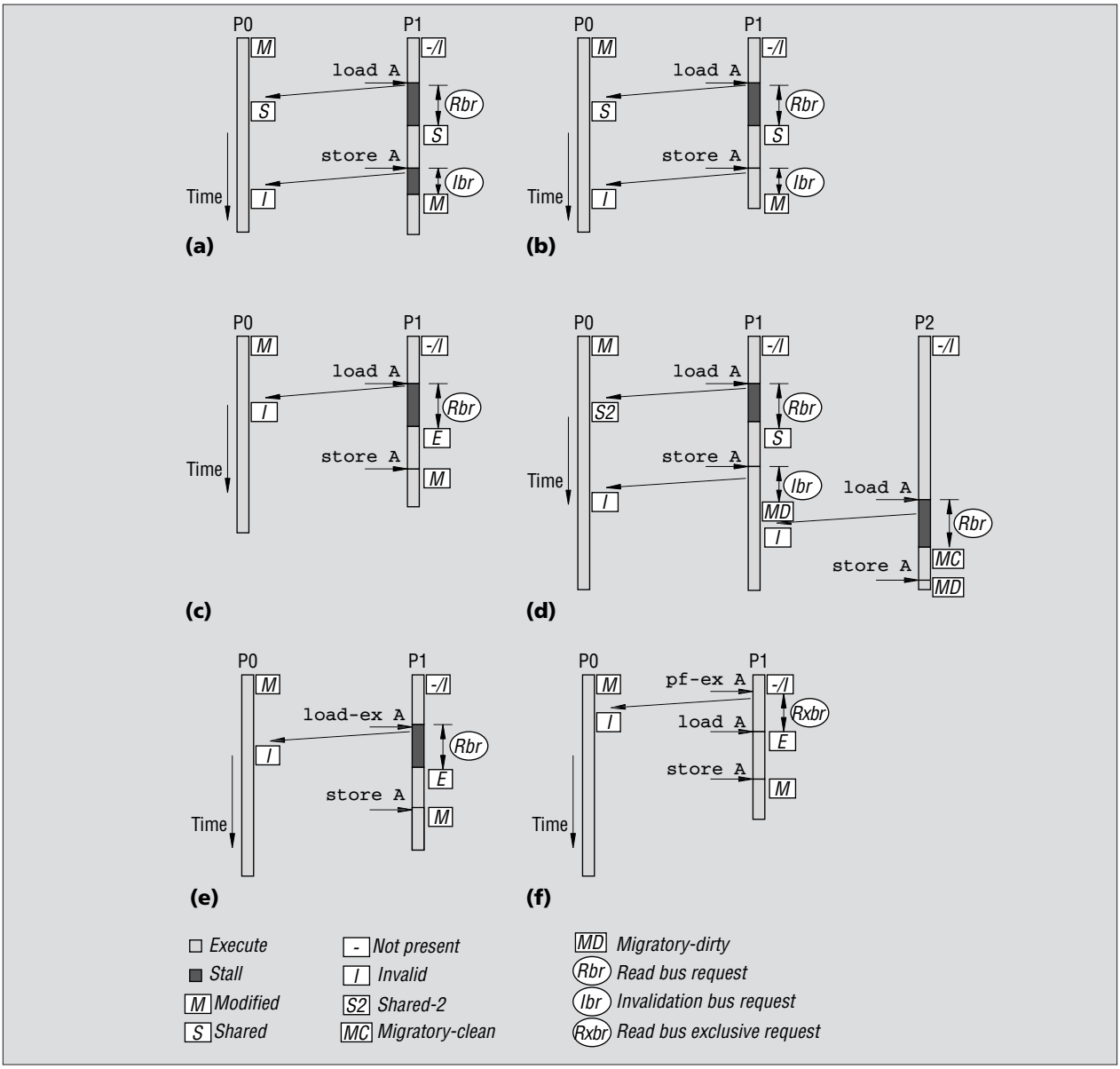

Figure 2. Invalidation bus traffic reduction techniques: (a) base under sequential consistency, (b) base under release consistency, (c) migrate-on-dirty, (d) adaptive detection scheme under release consistency, (e) load-exclusive instruction, and (f) exclusive prefetching.

SMPs.[10] Their simulation was based on traces collected from a real parallel machine with 12 processors. To augment each reference resulting in a miss, they inserted a prefetch instruction in the instruction stream some distance ahead of that reference. The modeled architecture had 32 Kbytes of direct-mapped private caches with a 32-byte block size and the Illinois coherency protocol, a 16-deep prefetch buffer, and a split-transaction bus with a round-robin arbitration scheme.

Surprisingly, the study found ineffective cache prefetching despite an assumed high memory latency of 100 processor cycles. For several architecture variations, speedups for five parallel programs were no greater than 39%, and degradations were as high as 7%. The following describes the main reasons for this:

- Prefetching increases bus traffic, which can result in performance degradation due to the bus-based architecture's strong sensitivity to bus traffic changes.
- Cache prefetching results in code expansion from both the prefetch instructions and code restructuring techniques used to support prefetch algorithms.
- Cache prefetching can adversely affect data sharing, especially in cases where prefetching initiates too early and another processor's write invalidates a prefetched block before it is used.
- Current prefetching algorithms are not very effective in predicting coherence misses. Actually, cache misses caused by data sharing represent the biggest challenge for designers, especially as caches become larger and coherence misses dominate the performance of parallel programs.

In their follow-up article, Tullsen and Eggers explored reducing these problems using architectural techniques and heuristics such as victim caching, compiler-based shared-data restructuring, and special prefetch algorithms for shared data, but limited effectiveness of cache prefetching in bus-based SMPs remains an issue.[11]

Read snarfing

Read snarfing is based on the heuristic that all invalid blocks will be needed in the future.[12] As shown in Figure 1c, a data block transferred on the bus as a read response not only updates the node that requested it but also updates all other caches that have the block in the invalid state. Typically, one processor writes to a block, and a large number of processors read from it. Here, only one read request will be transferred on the bus. Read snarfing is hardware-based and adds negligible extra complexity to the cache snoop mechanism.

Craig Anderson and Jean-Loup Baer explored read snarfing in an SMP with the Illinois protocol using an instruction-level simulator and six applications with different reference behaviors.[13] The simulated architecture had

- two-way set-associative 128-Kbyte private caches with dual-ported tag directories;
- a 64-bit bus width;
- an assumed 10-cycle main-memory latency;
- either an 8- or 64-byte block size; and
- 1, 4, 16, and 32 processors.

As the principal metrics, they used the speedup, the number of data bytes transferred during execution, and the number of bus transactions. They found that read snarfing improves performance by 30% to 67% for 32 processors with 64-byte cache blocks for three applications. For 8-byte cache blocks, performance improves up to 10%. Also, read snarfing reduces the number of bytes transferred —by up to 67% for 64-byte cache blocks —and reduces the number of bus transactions—by up to 70%. As the number of processors increases, read snarfing's effectiveness increases; it is poor in systems with less than 16 processors.

Read snarfing's main drawback is that its effectiveness highly depends on cache size. When cache size is relatively small, invalid cache blocks will probably be displaced from the cache, so read snarfing is not applicable. Although read snarfing is applicable when there are multiple data consumers, it is not applicable for other data sharing patterns, such as migratory sharing and 1P-1C (1 producer-1 consumer) situations. If there is dynamic change in a block's sharing pattern, read snarfing can hurt performance.

Software-controlled updating

Software-controlled updating further improves read snarfing. With this technique, a data producer initiates a block-update bus request by executing an update instruction after data production is finished. During this bus request, as shown in Figure 1d, all caches holding an invalid block copy are updated. This technique assumes the existence of an update instruction, which carries a block address and generates a block update transaction on the bus if the block is modified but does nothing if the block is not modified.

Jonas Skeppstedt and Per Stenstrom proposed a compiler algorithm that uses classic dataflow analysis techniques to insert update instructions where each modified memory block is written for the last time.[14] Instruction overhead due to inserted update instructions can be almost completely eliminated by replacing a store with a store-update instruction, which first performs a store and then an update. When the static insertion of an update instruction is successful, coherence misses and traffic can be reduced if other processors subsequently access the block.

However, if the block is not actively shared, update transactions waste bus traffic and increase the processor read stall time under sequential consistency. To mitigate this problem, Fredrik Dahlgren and his associates proposed a dynamic heuristic for detecting actively shared memory blocks.[15] If a memory module provides a block fetched into the cache on a load miss, the block is considered effectively private in the cache and an additional cache block status bit, denoted as a private flag, is set. For such blocks, update (or store-update) instructions have no effect.

The second problem is when the processor that caused the block update

modifies the block again, with no intervening access by another processor. To mitigate this problem, update requests can be delayed by temporarily inserting them in the write buffer or in an additional update buffer instead of immediately updating the block to other caches. Thus, if the processor continues to write to the block, it is still dirty in the cache and the update buffer is flushed at the synchronization point.

In an experimental analysis of software-controlled updating based on an execution-driven simulation, Dahlgren, Skeppstedt, and Stenstrom assumed a bus-based SMP with eight processors connected by a split-transaction bus.[15] Each processor had

- its own private 4-Kbyte direct-mapped write-through first-level cache,
- an infinitely large write-back second-level cache, and
- write-back buffers that allow use of release consistency potentials.

The cache coherence protocol was Illinois, cache line size was 32 bytes, and data bus width was 64 bits.

They compared the number of read misses and bus traffic for the base system, for the system with read snarfing, and for the system with software-controlled updating. They found that software-controlled updating eliminated almost all coherence misses for three of four considered applications, but read snarfing provided relatively small improvements. However, bus traffic increased for all applications from 4% to 108%. Bus traffic can be considerably reduced by buffering update instructions in a separate update buffer, but in the three applications, it was still higher than in the base system.

Unlike read snarfing, software-controlled updating reduces the number of read misses for blocks that exhibit a 1P-1C sharing pattern. However, software-controlled updating is based on read snarfing. Small caches and dynamic sharing-pattern changes can limit this approach's effectiveness, and useless update bus requests increase bus traffic.

Cache injection

Cache injection tries to overcome some of the other techniques' shortcomings, such as

- the minor effectiveness of read snarfing and software-controlled updating in SMPs with relatively small private caches,
- the high contention on the bus in cache prefetching and software-controlled updating, and
- the minor effectiveness of cache prefetching in reducing coherence misses.

In cache injection,[16] a consumer predicts its future needs for shared data by executing an openWin instruction, which does not initiate any bus transaction but only stores the first and last addresses of a range of consecutive cache blocks—an address window—in a special local injection table. There are two main scenarios when cache injection could happen: during the bus read transaction—injection on first read—or during the software-initiated update bus transaction—injection on update.

Injection on first read is always applicable when there is more than one consumer—for read-only shared data or for a 1P-MC (1 producer-multiple consumers) sharing pattern. Each consumer initializes its local injection table according to its future needs. When the first consumer executes a load instruction specifying the shared data, it sees a cache miss and initiates a read bus transaction. During this transaction, each cache controller snoops the bus, and if there is an injection hit, the cache controller stores the block into its cache, as shown in Figure 1e. For multiple consumers, only one read bus transaction is needed to update all consumers if they all have initialized their injection tables before this transaction.

Injection on update is applicable when shared data exhibit 1P-1C, 1P-MC, or migratory sharing patterns. In these scenarios, each consumer also initializes its injection table. After data production, the producer initiates an update bus transaction by executing an update instruction. During this transaction, all consumers snoop the bus, and if they find an injection hit, they catch the data block from the data bus and store it into their caches, as shown in Figure 1f. Cache injection may result in cache replacement if there is a conflict between the injected data and the current working set in the cache.

Hardware support for cache injection includes an injection table, proposed instructions, and a negligible modification of the bus control unit. The injection table is implemented as part of the cache controller. Each entry includes two address fields—*Laddr* and *Haddr*, which respectively define the first and last addresses of an address window—and a valid bit *V*. The proposed openWin instruction initializes an entry in the injection table by setting the valid bit and putting *Laddr* and *Haddr* values in the corresponding entry fields. If only one cache block should be injected, *Laddr* equals *Haddr*.

Because an entry's initialization requires two address fields, they can be created with two separate instructions—openWinL, which initialize *Laddr*, and openWinH, which initialize *Haddr*—or by using an instruction which specifies *Laddr* and the number of consecutive cache blocks. Instruction closeWin checks the injection table, and if there is an open window with a specified *Laddr*, it closes that window by resetting the valid bit. The compiler and/or programmer are responsible for inserting openWin and closeWin instructions at the consumer side and for inserting update and store-update instructions at the producer side.

Performance analysis of cache injection shows that this scheme, compared to read snarfing and software-controlled updating, can further cut the number of read misses and consequently bus traffic, as explained in the sidebar "Performance evaluation of cache injection."

Migrate-on-dirty

Migrate-on-dirty acquires an exclusive copy of the block when the block is exclusive or modified in another cache. Thus, copying the block to P1 and invalidating P0's copy are done in one bus

Performance evaluation of cache injection

We compared the number of read misses and the amount of bus traffic for the base system, the system with read snarfing, the system with software-controlled updating, and the system with cache injection using Limes, a tool for program-driven shared-memory multiprocessor simulations. For the workload, we used four parallel test applications well suited for demonstrating various data-sharing patterns—PC (1 producer–multiple consumers), MM (read only, multiple consumers), MS (migratory sharing), and Jacobi (1 producer–1 consumer)—and four applications from the Splash-2 suite—Radix, FFT, LU, and Ocean.[1] They are all written in C using ANL macros to express parallelism and compiled by the Gnu C compiler with the optimization flag –O2. We hand-inserted proposed instructions for software-controlled updating and cache injection support into the applications.

The modeled architecture is a bus-based SMP with 16 processors that uses the Illinois write-back invalidate cache coherence protocol. The bus supports split transactions and uses a round-robin arbitration scheme. We assumed a single-issue, in-order processor model with blocking reads. Processors execute a single cycle per instruction. Each processor includes only first-level cache memory. Instructions always hit into the cache, and a cache hit is solved without penalty. The relevant system parameters are a 32-byte cache line size, 64-bit data bus width, 2-pclk (processor clock) snoop cycle, 32-byte write-back buffer size, and an assumed 20-pclk memory latency. We used a 128-entry injection table in the evaluation.

Figure A shows the number of read misses and the bus traffic for parallel applications, normalized to the base system, when the caches are relatively small (left) and relatively large (right). For all applications, cache injection outperforms read snarfing and software-controlled updating, except for Jacobi, where software-controlled updating performs as well as cache injection. The effectiveness of cache injection relative to read snarfing and software-controlled updating is higher in the system with relatively small caches. Invalid blocks are frequently displaced from the cache, in which case snarfing is not applicable (applications PC, Radix, and Ocean). Next, cache injection can be effective in reducing cold misses when there are multiple consumers of shared data (applications MM and LU), but read snarfing can eliminate only coherence misses. Last, software control of the time window during which a block can be injected provides flexibility and adaptivity to different data-sharing patterns (applications MS and FFT).

Additional experiments, which varied architectural parameters, showed that cache-injection efficiency increases with the number of processors in the system, with cache memory size, and with memory latency. When the number of processors increases, the percentage of shared data and the number of sharers increase. Hence, the benefit of injection increases due to lowering the overall miss rate and reducing the bus traffic. Larger caches reduce the probability of collision between the injected data and the current working set. If the memory read cycle time is longer, we gain more by reducing the read stall time.

Reference

1. S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, IEEE Computer Soc. Press, Los Alamitos, Calif.,1995, pp. 24–36.

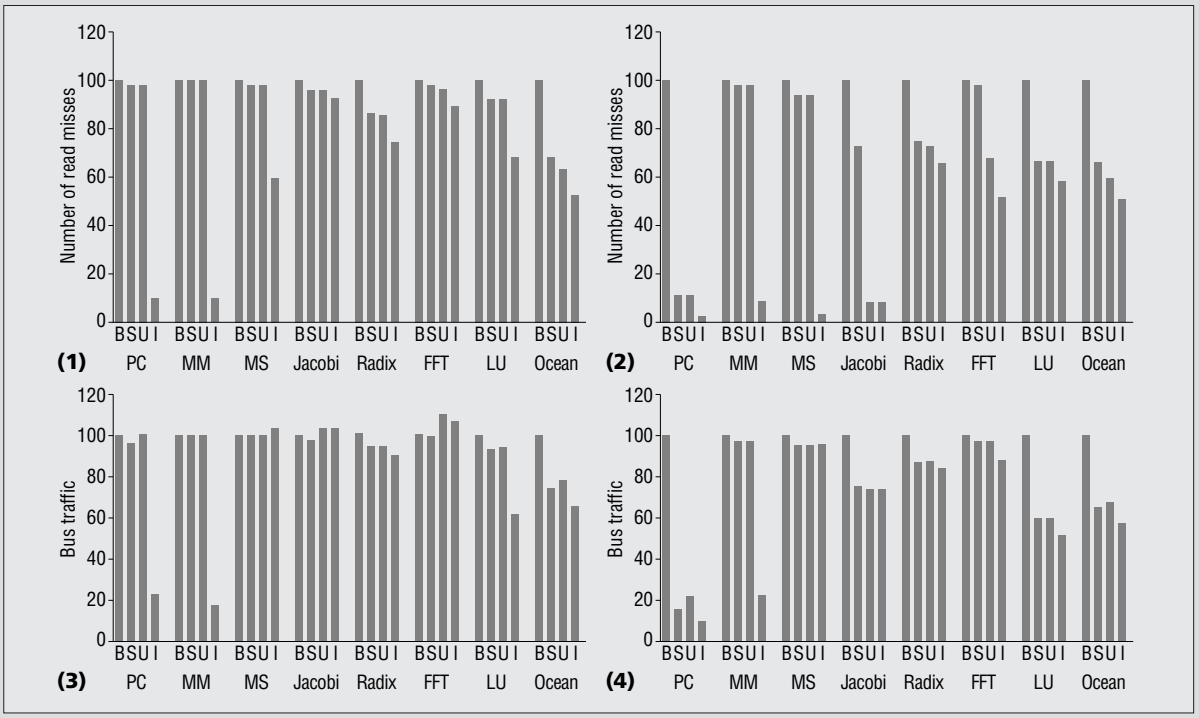

Figure A. Number of read misses (1–2) and bus traffic (3–4) relative to the base system. Cache_size = 64/128 Kbytes (128 Kbytes for FFT, LU, Ocean) (1, 3), and Cache_size = 1024 Kbytes (2, 4). (B = base system; S = read snarfing; U = software-controlled updating; I = cache injection.)

transaction, rather than two, as shown in Figure 2c. If the first access from P1 is a write, both base protocol and migrate-on-dirty incur the same overhead because they handle write misses in the same way. However, it is clear that this approach can increase the number of read misses and consequently bus traffic for sharing patterns other than migratory sharing.

An experimental study by Dahlgren and his associates showed that for applications in which migratory sharing prevails a migrate-on-dirty policy reduces bus traffic in two out of four considered applications by 25% and 33%.[15] However, in two other applications having a producer-consumer sharing pattern, migrate-on-dirty heavily increases the number of read misses—up to five times—and bus traffic—up to three times.

Adaptive migratory detection

Although the base write-invalidate protocol, based on a replicate-on-read-miss policy, is not appropriate for migratory data, a migrate-on-read-miss policy performs poorly for other data-sharing patterns. It motivated Alan Cox and Robert Fowler to propose an adaptive migratory detection scheme in which the two policies coexist.[17] In this scheme, a block is classified as migratory if one of the following conditions is true:

- one processor has written to the block, and thus made it dirty, and another processor first reads and then writes to it or
- one processor has written to the block, and thus made it dirty, and another processor writes to it.

If a block is classified as migratory, the adaptive protocol expects that the block will be modified at every processor it visits. Thus, if the block is not modified before it moves to another processor, this shows that the block is not currently migratory.

To support an adaptive protocol, Cox and Fowler extended the Illinois protocol with three new states—migratory-clean, migratory-dirty, and shared-2—and a migratory line on the bus. A new shared-2 state means that no more than two copies of the block exist. The only transitions into shared-2 come from the exclusive and modified states in response to a read request from the bus, as shown in Figure 2d. On an invalidation request from the bus to a block in state shared-2, the block is invalidated and the migratory line activated, and then the writing node changes the state from shared to migratory-dirty. When another node reads this block, it will get an exclusive copy of the block, and because it has not yet written to it, the state of the block will be migratory-clean. When that node writes to the block, the state changes to migratory-dirty. The aggressive protocol also switches from the replicate-on-read-miss policy to the migrate-on-read-miss policy if a processor has a write miss to a block with a single cached copy in either the exclusive or modified states. However, if another processor requests the block while it is still in the migratory-clean state, it will be reclassified as not migratory.

Dahlgren and his associates showed that adaptive migratory detection reduces bus traffic for three out of four applications by 21% to 31%.[15] This technique proves almost as efficient as migrate-on-dirty in reducing invalidation bus traffic in applications where migratory sharing is dominant, but it is more robust because it does not generate many useless misses for other data-sharing patterns.

Load-exclusive instruction

Another technique assumes the existence of a load-exclusive instruction, which forces the cache to obtain an exclusive copy of the requested block if the load misses in the cache, as shown in Figure 2e. Skeppstedt and Stenstrom's compiler algorithm uses dataflow analysis at the intraprocedural level to recognize each load instruction, followed with a store instruction to the same address, and replaces the load instruction with a load-exclusive instruction.[18]

Dahlgren and his associates showed that this approach reduces bus traffic by 3% to 31%, and proves as effective as the previous two techniques in applications exhibiting significant migratory sharing.[15] This robust approach does not increase the number of read misses for nonmigratory sharing patterns.

Exclusive prefetching

In software-controlled cache prefetching, a prefetched migratory block is in the shared state. A write to the block causes the invalidation bus request and stalls the processor under sequential consistency. However, this can be avoided with exclusive prefetching of all the shared data that will be written, as shown in Figure 2f.

To support exclusive prefetching, a prefetch-exclusive instruction is needed. This instruction prefetches the data into the cache in exclusive mode, invalidating copies in other caches. Todd Mowry developed an extension of the base compiler algorithm for prefetching that supports exclusive prefetching.[8]

Dean Tullsen and Susan Eggers found that exclusive prefetching is ineffective at reducing execution time for four of the five applications.[11] The problem of unnecessary invalidate operations caused by prefetching write accesses was only evident in one application, where exclusive prefetching reduced execution time by 2%.

MODERN MICROPROCESSORS include support for software-controlled prefetching, but other techniques outlined in this article can further improve performance at minimal cost. Combining software-controlled cache prefetching and cache injection—which in a way encompasses the other two considered techniques, read snarfing and software-controlled updating—seems especially promising. Further combinations that incorporate invalidation bus traffic reduction techniques, such as exclusive prefetching and adaptive migratory detection, could significantly improve performance and allow bus-based SMPs with more processors. Much can be

gained from exploring the effectiveness of these combined techniques both in state-of-the-art and future bus-based SMPs.

References

1. D. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/ Software Approach,* Morgan Kaufmann, San Francisco, 1998.
2. M. Tomasevic and V. Milutinovic, *Tutorial on the Cache Coherence Problem in Shared-Memory Multiprocessors: Hardware Solutions,* IEEE Computer Society Press, Los Alamitos, Calif., 1993.
3. M. Tomasevic and V. Milutinovic, "Hardware Approaches to Cache Coherence in Shared Memory Multiprocessors, Part I," *IEEE Micro,* Vol. 14, No. 5, Oct. 1994, pp. 52–59.
4. S.J. Eggers and R.H. Katz, "Evaluating the Performance of Four Snooping Cache Coherence Protocols," *Proc. 16th Ann. Int'l Symp. Computer Architecture,* IEEE Computer Soc. Press, Los Alamitos, Calif., 1989, pp. 2–15.
5. V. Milutinovic, *Microprocessors and Multimicroprocessor Systems,* John Wiley & Sons, New York, 2000.
6. J. Protic, M. Tomasevic, and V. Milutinovic, *Distributed Shared Memory: Concepts and Systems,* IEEE Computer Soc. Press, Los Alamitos, Calif., 1998.
7. F. Dahlgren, M. Dubois, and P. Stenstrom, "Sequential Hardware Prefetching in Shared Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* Vol. 6, No. 7, July 1995, pp. 733–746.
8. T. Mowry, *Tolerating Latency through Software-Controlled Data Prefetching,* doctoral dissertation, Computer Science Dept., Stanford University, Stanford, Calif., 1994.
9. T. Mowry and C. Luk, "Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling," *Proc. 30th Ann. Symp. Microarchitecture,* IEEE Computer Soc. Press, Los Alamitos, Calif., 1997, pp. 314–320.
10. D. Tullsen and S. Eggers, "Limitations on Cache Prefetching on a Bus-Based Multiprocessor," *Proc. 20th Ann. Int'l Symp. Computer Architecture,* IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp. 392–403.
11. D. Tullsen and S. Eggers, "Effective Cache Prefetching on Bus-Based Multiprocessors," *ACM Trans. Computer Systems,* Vol. 13, No. 1, Feb. 1995, pp. 57–88.
12. L. Rudolph and Z. Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *Proc. 11th Ann. Int'l Symp. Computer Architecture,* IEEE Computer Soc. Press, Los Alamitos, Calif., 1984, pp. 340–347.
13. C. Anderson and J.-L. Baer, "Two Techniques for Improving Performance on Bus-Based Multiprocessors," *Proc. First Int'l Symp. High Performance Computer Architecture,* IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp. 256–275.
14. J. Skeppstedt and P. Stenstrom, "A Compiler Algorithm that Reduces Read Latency in Ownership-Based Cache Coherence Protocols," *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp. 69–78.
15. F. Dahlgren, J. Skeppstedt, and P. Stenstrom, "Effectiveness of Hardware-Based and Compiler-Controlled Snooping Cache Protocol Extensions," *Proc. High-Performance Computing,* Springer-Verlag, Berlin, 1995, pp. 87–92.
16. A. Milenkovic and V. Milutinovic, "Cache Injection on Bus-Based Multiprocessors," *Proc. Workshop on Advances in Parallel and Distributed Systems,* IEEE Computer Soc. Press, Los Alamitos, Calif., 1998.
17. A.L. Cox and R.J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," *Proc. 20th Ann. Int'l Symp. Computer Architecture,* IEEE Computer Soc. Press, Los Alamitos, Calif., 1993, pp. 98–108.
18. J. Skeppstedt and P. Stenstrom, "A Compiler Algorithm to Reduce Ownership Overhead in Cache Coherence Protocols," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM Press, New York, 1994, pp. 286–296.

Aleksandar Milenkovic is an assistant professor in the Department of Computer Engineering, School of Electrical Engineering, at the University of Belgrade. His research interests include computer architecture, parallel computer systems, computer-aided design, and the Internet. He received a BS, an MS, and a PhD in computer engineering from the University of Belgrade. He is a member of the IEEE and IEEE Computer Society. Contact him at milenkovic@computer.org.

How to Reach IEEE Concurrency

Writers

For detailed information on submitting articles, write for our editorial guidelines (concurrency@computer.org), or access computer.org/concurrency/edguide.htm.

Letters to the Editor

Send letters to

Shani Murray
IEEE Concurrency
10662 Los Vaqueros Circle
Los Alamitos, CA 90720

Please provide an e-mail address or daytime phone number with your letter.

Subscription Change of Address

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Concurrency*.

Membership Change of Address

Send change-of-address requests for the membership directory to directory.updates@computer.org.

Missing or Damaged Copies

If you are missing an issue or you received a damaged copy, contact membership@computer.org.

Reprints of Articles

For price information or to order reprints, send e-mail to concurrency@computer.org or fax +1 714 821 4010.

Reprint Permission

To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and trademarks Manager, at whagen@ieee.org.