Reducing Coherence Overhead in Shared-Bus Multiprocessors

Sangyeun Cho[1] and Gyungho Lee[2]

[1] Dept. of Computer Science
[2] Dept. of Electrical Engineering
University of Minnesota, Minneapolis, MN 55455, USA
E-mail: {sycho@cs,ghlee@ee}.umn.edu

Abstract. To reduce the overhead of cache coherence enforcement in shared-bus multiprocessors, we propose a self-invalidation technique as an extension to write-invalidate protocols. The technique speculatively identifies cache blocks to be invalidated and dynamically determines when to invalidate them locally. We also consider enhancing our self-invalidation scheme by incorporating read snarfing, to reduce the cache misses due to incorrect prediction. We evaluate our self-invalidation scheme by simulating SPLASH-2 benchmark programs that exhibit various reference patterns, under a realistic shared-bus multiprocessor model. We discuss the effectiveness and hardware complexity of self-invalidation and its enhancement with read snarfing in our extended protocol.

1 Introduction

There are at least three types of overhead in enforcing cache coherence in a shared-bus multiprocessor using a write-invalidate snooping protocol [8, 5]. Firstly, the bus bandwidth is consumed, i.e. invalidations are generated when a processor writes to a shared block. Secondly, the machine experiences *coherence misses*. A coherence miss is a miss due to prior invalidation of the cache block by a remote processor. Thirdly, coherence actions need to look up the cache state and tag memory and possibly update it, which can increase the contention with the local processor. A duplicated state and tag memory can minimize the contention. With large caches employed in modern shared-bus multiprocessors, the coherence overhead is a major performance degrading factor, and reducing invalidations is extremely important.

In this paper, we propose a *self-invalidation* technique to reduce the invalidation traffic. Self-invalidation is a technique to invalidate cache blocks locally without an explicit invalidation message. Our scheme is a simple hardware-oriented extension to write-invalidate protocols and can be easily adopted in current multiprocessor technology with little hardware complexity. Moreover, it does not change the memory model seen by a programmer. We complement the self-invalidation scheme with a variant of *read snarfing* [1] to reduce the cache misses due to incorrect prediction. Read snarfing has been discussed and evaluated as an enhancement to snooping cache coherence protocols that takes advantage of the inherent broadcasting nature of the bus [9, 4, 1].

State	Event	Request	Reply		New State
			SI*	Shared*	
I	Rmiss	Rreq	0	0	E
	Rmiss	Rreq	1	0	E+
	Rmiss	Rreq	0	1	S
	Rmiss	Rreq	1	1	S+
	Wmiss	Wreq	0	0	M
	Wmiss	Wreq	1	0	M+
S	Read	-	-		no change
	Write	Ireq	-		M
S+	Read	-	-		no change
	Write	Ireq	-		M+
	Sync	-	-		I
E	Read	-	-		no change
	Write	-	-		M
E+	Read	-	-		no change
	Write	-	-		M+
M	Read	-	-		no change
	Write	-	-		no change
M+	Read	-	-		no change
	Write	-	-		no change

(a) Local events

State	Request	New State	Assert		Data
			SI*	Shared*	
S	Rreq	no change	0	1	-
	Wreq	I	1	0	-
	Ireq	I	0	0	-
S+	Rreq	no change	1	1	-
	Wreq	I	1	0	-
	Ireq	I	0	0	-
E	Rreq	S	0	1	-
	Wreq	I	1	0	-
E+	Rreq	S+	1	1	-
	Wreq	I	1	0	-
M	Rreq	S+	1	1	Provide
	Wreq	I	1	0	Provide
M+	Rreq	I	1	0	Provide
	Wreq	I	1	0	Provide

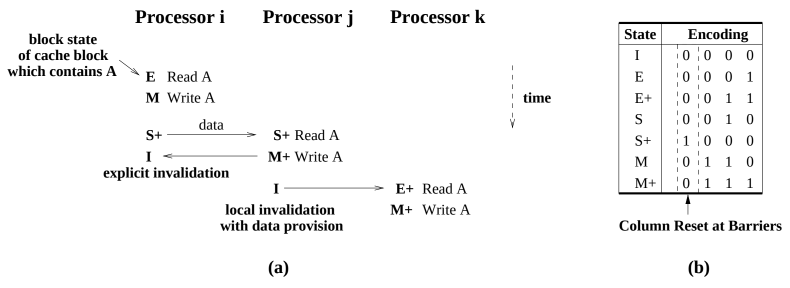
(b) Bus-induced actions

Fig. 1. State transition table for a self-invalidation technique

2 A Self-Invalidation Technique

The technique we propose has two phases, *marking* and *local invalidation*.
Marking. Cache blocks are marked for self-invalidation on specific events. This marking is incorporated in cache block states as can be seen in Fig. 1, where the extended protocol is shown. In the figure, `SI*` and `Shared*` are special bus lines that distribute sharing information during bus transactions, the former of which is introduced in this section and the latter of which is the "shared" line used in many other shared-bus multiprocessors. The states in bold are from the Illinois protocol (**M**odified, **E**xclusive, **S**hared, and **I**nvalid) and others (**M+**, **E+**, and **S+**) mark a cache block for self-invalidation.

Cache blocks which have been written in a shared state are marked for eliminating future invalidations. A typical example of such blocks is the one having a migratory object. A migratory object is in many cases guarded by lock-unlock synchronizations and exclusive access is observed, giving us an opportunity for eliminating explicit invalidations. Our marking will detect migratory objects like Cox and Fowler's adaptive protocol [3] does; an example of such a case is shown in Fig. 2 (a). Unlike their scheme which pays attention to strictly migratory objects, however, we aggressively classify shared blocks that have a history of being written as self-invalidation candidates. This history is passed on to a requesting processor through `SI*` and `Shared*` lines when a read request is seen on the bus.
Local invalidation. A Cache block in **M+** state is locally invalidated when another processor requests the block. The requesting processor will have the block in **E+** state so that a later write will not generate an invalidation, and the state carries with it a speculation that the block will be locally invalidated as it becomes shared by multiple processors. Typical migratory objects will carry



State	Encoding			
I	0	0	0	0
E	0	0	0	1
E+	0	0	1	1
S	0	0	1	0
S+	1	0	0	0
M	0	1	1	0
M+	0	1	1	1

Fig. 2. An example of (a) a migratory block, and (b) state encoding

E+ or **M+** state as depicted in Fig. 2 (a). Cache blocks in **S+** state are all invalidated at a barrier synchronization point, since barriers give a good hint that the sharing pattern will change across them. Our experimental results with `Ocean` and `Radix` from the SPLASH-2 benchmark suite support this observation.

Implementation. For the marking phase of the scheme, a special bus line, `SI*`, is needed. `SI*` is asserted by the owner of a requested cache block, which gives a hint to the requesting node in deciding the state of the incoming block.

For the timing of local invalidation, the memory system should be able to see the synchronizations. This is not hard, and many optimizations such as release consistency [6] assume that lock, unlock, and barrier synchronizations are seen by the memory system. We consider barrier synchronizations in this paper. To efficiently perform local invalidation at barriers, a column-reset circuit and a redundant encoding of cache block states are needed. An example of such an encoding is shown in Fig. 2 (b). Although 3 bits suffice to encode 7 states, we add 1 more bit for efficient state change at barriers. Resetting this bit column will force all blocks in **S+** state into **I** state at the same time. With a 40-bit address and 32-Byte block size, a four-way set-associative 1-MB cache will have an additional 0.714% ($\frac{2}{32\times 8+(40-5-13)+2)}$) of memory overhead, compared to a 2-bit encoding of 4 states.

Enhancing with read snarfing. We investigate the potential of combining read snarfing with our self-invalidation scheme. The two techniques have contradicting properties in that self-invalidation tries to keep the degree of sharing low for shared cache blocks, whereas snarfing tries to maximize the sharing at all read requests. Read snarfing hence cannot directly mix with the self-invalidation technique. We modify read snarfing so that processors will read-snarf only when a read request results in a block cached in at least two caches. In other words, read snarfing becomes conservative, and it waits until the second read miss occurs for the same cache block without an intervening write. This is illustrated in Fig. 3. The rationale behind the conservative read snarfing is the observation made by Eggers and Katz [4] that most invalidation misses are caused by a re-read by a single processor. For migratory objects, the conservative read snarfing does not hinder self-invalidation. The conservative read snarfing can be easily implemented by using `Shared*` line as a hint for snarfing. If the line is not

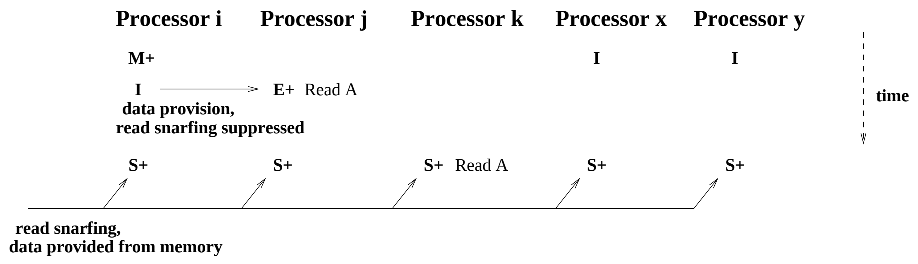


Fig. 3. An example of conservative read snarfing

asserted, snarfing action is simply suppressed, and if the line is asserted, read snarfing is performed and the resulting state will be **S+** for all cached copies.
Discussion. Cox and Fowler's scheme [3] identifies migratory objects at run time and tries to eliminate invalidations for such objects. Once a cache block is deemed to be migratory, further read requests will obtain the block in an exclusive state. Compiler algorithms to identify and optimize for the migratory sharing pattern were also studied by Skeppstedt and Stenström [10]. They use a data-flow analysis to recognize uninterrupted *read-modify-write* sequences to the same address. *Dynamic Self-Invalidation* techniques were studied by Lebeck and Wood [7]. They identify blocks for self-invalidation by associating a version number with each block in the cache and with each entry in the directory. On responding to a read request, the directory compares the version number of the entry with that of the cache block, and provides the block with a marking if the two numbers mismatch. Since they assumed a directory-based coherence protocol, their technique needs a special sequencing circuit to regenerate the cache block addresses and send acknowledgements at each synchronization point.

3 Performance

Experimental setup. We use a program-driven simulation to simulate a shared-bus multiprocessor model with 16 processors. Our simulator consists of two parts: a front end, MINT [11], which simulates the execution of the processors, and a back end that simulates the memory system and the bus. The shared bus is closely modeled after the POWERpath-2 bus [5]. It is clocked at 50 MHz and has a 40-bit address bus and a 64-bit data bus, which are arbitrated independently. Each bus transaction consumes five bus cycles, and a cache block can be transferred in one bus transaction. Our simulator is capable of capturing contention within the bus and in the state and tag memory due to conflicting accesses from the processor and the bus. A processor node contains a high-performance microprocessor and two levels of data caches: a small direct-mapped first-level (L1) cache and a set-associative second-level (L2) cache. The block size of the caches is 32 Bytes. Write-through policy is used between the L1 and L2 caches, and write-back policy is used between the L2 caches and the main memory. We

Program	Description	Input
`Cholesky`	Cholesky factorization of a sparse matrix	`bcsstk14`
`LU`	LU decomposition of a dense matrix	300×300 matrix, 16×16 block
`FFT`	Complex 1-D version of radix-$\sqrt{N}$ six-step FFT	256 K points
`Ocean`	Ocean basin simulator	130×130 ocean, 10^{-7} tolerance
`Radix`	Radix sorting	200,000 integers
`Water`	Simulates evolution of a system of Water molecules	512 molecules, 3 time steps

Table 1. Summary of benchmark programs

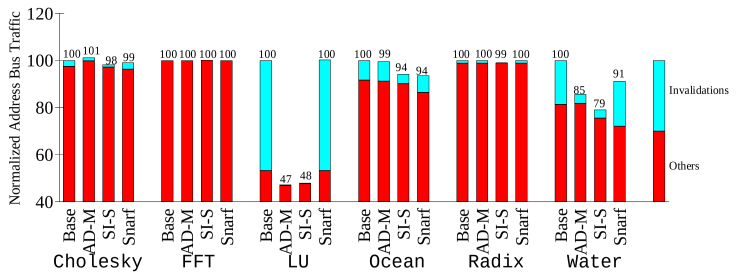


Fig. 4. Bus traffic (*Base*: base Illinois, *AD-M*: adaptive protocol for migratory object [3], *SI-S*: self-invalidation with conservative snarfing, *Snarf*: snarfing)

assume no stalls for instruction fetching, and an instruction can be executed in a processor clock cycle (pclock). The L1 cache is 2 KBytes and the access time is hidden if an access hits in the cache. It has a 4-pclock block fill time. The 4-way set-associative L2 cache is 128 KBytes with a 40-ns cycle time. We assume a fully interleaved main memory with an access time of 120 ns.

We use 6 programs from the SPLASH-2 benchmark suite [12] to drive our simulator. Program descriptions and inputs are summarized in Table 1. The `LU` program we use is `LU-C` (contiguous block allocation version). For measurement, we gather statistics during the parallel sections only.

Traffic reduction.[3] Fig. 4 shows the bus traffic of 4 different schemes. As can be seen, both *AD-M* and *SI-S* schemes could reduce the invlidations considerably. In `LU`, invalidations were eliminated almost completely. Two programs, however, exhibit some difference between the two schemes: `Ocean` and `Radix`. `Ocean` is known to have little migratory sharing, and it has a lot of barrier synchronizations [12]. `Radix` also contains more barriers than locks. *AD-M* did not reduce much traffic in these two programs due to few strictly migratory data accesses, whereas *SI-S* reduced a large portion, by invalidating marked cache blocks at barriers. Notice that *Snarf* did not reduce coherence traffic, although it reduced other traffic (mostly memory requests) in `Ocean` and `Water`. In `Cholesky`, *AD-M* and *SI-S* generated more memory request traffic than *Base*, due to a slight increase in the cache miss rate from incorrect prediction, which was not covered

[3] We focus only on the bus traffic in this paper. Detailed results including execution time are found in [2].

by the conservative read snarfing. On average, *AD-M*, *SI-S*, and *Snarf* reduced 11.1%, 13.6%, and 2.7% of the bus traffic respectively.

4 Summary

Considering the current trend of adopting large (multi-megabyte) caches, the coherence overhead becomes a more dominant factor degrading performance. Reducing the coherence overhead, which in turn reduces the bandwidth requirement on the bus, is very important for future shared-bus multiprocessors.

We propose a simple hardware-oriented self-invalidation technique to reduce the coherence traffic. Using a program-driven simulation of six programs from the SPLASH-2 benchmark suite, we observed a reduction of coherence traffic averaging 71.6%. We modified read snarfing to be combined with our self-invalidation scheme to reduce the cache misses. We observed that the combined scheme reduced the bus traffic by 13.6% on average, which promises potential improvement of execution time if coherence traffic dominates the bus traffic of a program. The proposed scheme adds little to the hardware complexity.

References

1. Anderson, C. and Baer, J.-L.: Two Techniques for Improving Performance on Bus-Based Multiprocessors. Proc. of HPCA-1, pp. 256 – 275, Jan., 1995.
2. Cho, S. and Lee, G.: Reducing Coherence Overhead in Shared-Bus Multiprocessors, DICE Project TR No. 16, Dept. of Elec. Eng., Univ. of Minn., Feb. 1996.
3. Cox, A. and Fowler, R.: Adaptive Cache Coherency for Detecting Migratory Shared Data, Proc. of 20th ISCA, pp. 98 – 107, May, 1993.
4. Eggers, S. J. and Katz, R. H.: Evaluating the Performance of Four Snooping Cache Coherency Protocols, Proc. of 16th ISCA, pp. 2 – 15, June, 1989.
5. Galles, M. and Williams, E.: Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor, Proc. of 27th Hawaii Int'l Conf. on System Sci., Vol. 1, pp. 134 – 143, 1994.
6. Gharachorloo, K., *et al.*: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, Proc. of 17th ISCA, pp. 15 – 26, June, 1990.
7. Lebeck, A. and Wood, D.: Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors, Proc. of 22nd ISCA, June, 1995.
8. Lovett, T. and Thakkar, S.: The Symmetry Multiprocessor System, Proc. of 17th ICPP, pp. 303 – 310, Aug., 1988.
9. Rudolph, L. and Segall, Z.: Dynamic Decentralized Cache Schemes for MIMD Parallel Processors, Proc. of 11th ISCA, pp. 340 – 347, 1984.
10. Skeppstedt, J. and Stenström P.: Simple Compiler Algorithms to Reduce Ownership Overhead in Cache Coherence Protocols, Proc. of 6th ASPLOS, pp. 286 – 296, Oct., 1994.
11. Veenstra, J. and Fowler, R.: Mint: A Front-End for Efficient Simulation of Shared-Memory Multiprocessors, Proc. of 2nd MASCOTS, Jan. – Feb., 1994.
12. Woo, S., *et al.*: The SPLASH-2 Programs: Characterization and Methodological Considerations, Proc. of 22nd ISCA, pp. 24 – 36, June, 1995.

This article was processed using the LaTeX macro package with LLNCS style