Two-Phase Write Posting on Symmetric Multiprocessors

Byung-Kwon Chung
Yongjoon Lee
Sun Microsystems
901 San Antonio Road
Palo Alto, CA 94303

Jih-Kwon Peir
CISE Department
University of Florida
Gainesville, FL 32611

Konrad Lai
MRL, Intel Corp.
5200 NE Elam Young Parkway
Hillsboro, OR 97124

Abstract *Cache coherence activities with write-invalidate protocol in Symmetric Multiprocessors not only incur overhead but may increase cache miss ratios due to unnecessary invalidations. Under software synchronization models, a lazy cache coherence protocol delays write invalidations and permits inconsistent copies of the same cache line existing in different caches. In this paper, we propose a demand-driven two-phase deferred cache coherence model which further delays writes to be observed by other processors until a processor requests the new data after certain synchronization instructions. Data dependence can be maintained by identifying when the new data must be fetched and reconciled. Cycle-by-cycle execution-driven simulation of SPLASH-2 workload shows that the two-phase deferred coherence protocol can out-perform the eager protocol up to 30% for some workload.*

Keywords: Symmetric Multiprocessor (SMP), Cache Coherence Protocol, Snooping Bus, False Sharing, Execution-Driven Simulation

1 Introduction

Symmetric multiprocessor (SMP) technology has been adapted in most today's severs to achieve higher performance. Each processor is typically equipped with a coherent cache memory to hide the memory access latency and to reduce the critical shared bus traffic. The key cache coherence function is to guarantee that each processor always observes all the memory writes from any other processor. Whenever a processor writes to a memory location, the old copy of the data existing in other caches must be updated or invalidated. Such a cache coherence activity not only incurs heavy overhead, but may increase cache miss ratios due to the false-sharing behavior. The false-sharing occurs when multiple processors attempt to update different portion of a cache line about the same time [5, 4].

In fact, this cache coherence activity can be deferred until the next synchronization instruction to allow multiple processors updating the same cache line as long as software maintains an order of reads and writes to each memory location [1, 6, 3]. Under the software model, synchronization instructions are inserted to enforce certain order of memory accesses from different processors. As a result, any producer/consumer data among the processors must be synchronized. This software synchronization model allows reordering memory reads and writes without violating sequential consistency [1, 6]. It provides flexibility to cache coherence mechanisms on when to make writes to be globally visible.

The traditional *eager* coherence protocol enforces the cache coherence on every memory write [10]. The *lazy* coherence protocol, on the other hand, permits temporary inconsistent cache copies [3, 8, 9, 2, 7]. Those copies of data in different caches need to be reconciled only after the execution of a synchroniza-

tion instruction. Although this lazy coherence protocol eliminates unnecessary cache coherence activities, it incurs significant overhead on posting writes and reconciling cache lines at each synchronization instruction.

In this paper, we describe a two-phase *deferred* cache coherence model on bus-based SMPs that further delay posting writes until a processor requests the new data. There are two fundamental issues for maintaining correct data dependence under lazy coherence protocols. The first issue is when each processor must recognize that it is in danger to access stale data. The second issue is when the inconsistent data created by multiple writers must be reconciled and posted. The proposed two-phase deferred coherence protocol separates the event of *write notification*, and the event of *data reconciliation and posting*. At the first phase, all the cache lines are *marked* after each synchronization instruction notifying the need for reconciliation of certain inconsistent copies. At the second phase, a marked cache line when first referenced, triggers the reconciliation to merge the inconsistent lines into consistent state. This demand-driven data reconciliation and posting alleviate the overhead at each synchronization point. Furthermore, in case the inconsistent cache lines are replaced before they are referenced, the reconciliation can be overlapped with the normal execution.

An efficient hardware *merging* technique is used to minimize the overhead associated with reconciliation. This technique uses the original copy of the data in memory to identify and merge the modified portion of a cache line for restoring the consistent copy. Cycle-by-cycle execution-driven simulation of SPLASH-2 workload [12] shows that the two-phase deferred coherence protocol can out-perform the eager protocol up to 30% for some workload.

2 Motivation

In the popular Single-Program-Multiple-Data (SPMD) computation of parallel programs, the same program is sent to the participating processors and individual processor executes different pieces of work by operating on different pieces of data. A parallel program is laid out as a sequence of code regions separated by *barrier* synchronization primitives. At each barrier, no process is allowed to proceed until all the processes participating the barrier synchronization arrive so as to enforce proper data dependency. In addition, the other common synchronization requirement in SPMD is to allow mutual-exclusive updates of certain shared variables such as loop indices, processor IDs, etc. by individual processors during the course of parallel execution.

The barrier and the mutual exclusion are the two synchronization primitives used in the parallelized SPLASH-2 application suites [12]. The use of mutual exclusion and barrier synchronization presents two fundamental properties. First, a barrier must be inserted to enforce a producer/consumer data dependence between two processes. As a result, the updates of a shared variable only need to be observed by the consumer process after the execution of a barrier. Second, although the mutual-exclusive updates of a shared variable in a critical section can be executed in any order, the updates must be observed by other processes once the updating process leaves the critical section. Based on these observations, a *deferred* coherence protocol can be designed to postpone write posting until the new data is requested after the synchronization.

The proposed deferred coherence protocol works well with the release memory consistency model [6]. Given that software maintains proper order of memory reads and writes to the same memory location, hardware only needs to enforce the order of memory reads/writes with respect to synchronization instructions. As a result, writes can be deferred until after the subsequent synchronization point.

3 Two-Phase Deferred Coherence Protocol

In the proposed deferred coherence protocol, a new *partially-modified (P)* state is added

to the write-invalidate, write-back, Modified-Exclusive-Shared-Invalid (MESI) state protocol. A line in the P-state means the line is valid and has been modified by the local processor, meanwhile, such a line may also be valid and possibly modified in other caches. A S-state line is changed to the P-state upon a write-hit and the write can be performed locally. When a write-miss occurs, the new line is set to P-state if the line is also valid in another cache. In addition, an M-state line is changed to the P-state when another processor encounters a write-miss to the same line. In essence, the additional P state permits a line to be shared and modified to enable concurrent reads and writes to different portion of the same line among multiple processors.

Since a line in the P-state or even the S-state may be stale after each synchronization barrier, it is necessary to invalidate/reconcile those inconsistent lines. The reconciliation of the P-state lines can be further delayed until the line is first accessed or replaced after the barrier. However, each processor must identify potential stale lines existing in its local cache. We add one extra bit called the *mark bit* to each entry in the cache tag directory. All mark bits in every cache directory are *set* after each synchronization barrier to indicate potential stale lines as follows.

1. Marked P-state: The line must be reconciled upon the first access from either the local processor or the remote processor through the snooping bus. The request is reissued afterwards.

2. Marked M-state: The line is treated the same as a M-state line except the marked M-state line is sent to both the requester and the memory on the first access to update the memory copy.

3. Marked S-state: The line is treated the same as a S-state line if the line is not present in the P-state in any other cache. The line is invalid otherwise.

4. Marked E, I states: The same as the corresponding unmarked states.

The mark bits can be implemented as a separate array for fast set/reset all the bits. The corresponding mark bit is reset at the first access after a barrier to resume normal coherence activities to the cache line. Note that a marked S-state line become suspicious. The is because the S and the P states of the same line can coexist in multiple caches. Reconciliation of individual line is also required when both marked and unmarked P-state lines are replaced from the cache.

Requests from local processor:

1. *Read-hit:* When the requested line is found in the local cache, the data can be accessed and the state, M, E, S, or P remains unchanged.

2. *Write-hit:* The write can be performed locally without any global request. The states, S and E, are changed to P and M respectively, while states P and M remain unchanged.

3. *Read-miss:* A line-fill request is issued. The new state is E when the requested line is not present in any other cache. If the requested line is M in another cache, the cache which owns the modified copy supplies the data, and the new state is also set to E while the owner set to I state after writing back the line into memory. If the requested line exists in other caches in any other state, E, S, or P, the new state becomes S and the line is fetched from the memory.

4. *Write-miss:* Write-allocation is assumed. Upon a write-miss, the target line is fetched from the memory. The new state is M when the requested line does not exist in any other cache. Otherwise, the new state becomes P.

5. *Hit a marked P-state or replace a P- or marked P-state line:* Cache line reconciliation is initiated. Detailed description will be given in the next section.

Requests from snooping bus:

1. *Snooping read-hit:* Upon a snooping read-hit, the processor sends the modified (M-state) data to the requester and the memory. The line is invalidated afterwards. In case of a hit to an E-state, the state is changed to S without involving any data transfer. No action is taken when the state is either S, or P.

2. *Snooping write-hit:* Similar to a snooping read-hit except that no data transfer is needed for a M-state hit and the state is changed to P.

3. *Snooping read/write hits a marked line:* Follow the rules of accessing a marked line.

4. *Replacement write-back hit to a P- or marked P-state line:* Upon a hit, the snooping agent also writes back the P- or marked P-state line so that the reconciliation of the line can be carried out at the memory controller. The line is invalidated afterwards. The line in the S-state is simply invalidated without sending the data.

4 Reconcile Cache Lines

The performance of the deferred coherence protocol depends heavily on an efficient way of merging the partially modified lines. The write-back of a P- or marked P-state line is triggered under two conditions. First, a P- or marked P-state line is replaced from a cache. Second, a processor or a snooping request hits a marked P-state line.

When a partially-modified line write-back is initiated, all other processors that have the same P-state or marked P-state line also write back their own copy of the line. It is essential to provide an efficient way of merging those partially-modified cache line copies to restore the consistent copy in the memory. To eliminate the need of recording the portion of the cache line being modified in each processor, we use a *merging* technique to identify the modified portion and to reconcile the inconsistent

$$New\ Data = Old\ Data \oplus \left(\bigcup_{i=1}^{n} (Old\ Data) \oplus (Pstate\ Data)_i \right)$$

Figure 1: Reconciliation of n P-state copies

copies. This technique requires a special design of the memory controller. Upon receiving a partially-modified write-back request, the memory controller begins to merge the original cache line content stored in the memory with the new copies of the cache line from involved processors in a pipeline fashion. Basically, the merging algorithm performs a sequence of *exclusive-or* logic operations to identify the bit positions where the modification has been made. The new cache content can then be obtained by complementing the values of those bit positions as shown in Figure 1.

To simplify the memory controller design, we assume the multiple copies of the partially modified line arrive at memory controller continuously without being intervened by other requests. In order to satisfy this requirement, the processor from which the P-state write-back is initiated must hold the data bus until the completion of all the write-backs. In addition, the snooping controllers, once receiving a P-state write-back request, will preempt any other request. Since the P-state write-back involves multiple copies of a cache line and each copy must be sent sequentially through a shared data bus, the overhead associated with the merging operations can be overlapped with the P-state line transfer.

5 Performance Evaluation

Mint, an execution-driven MIPS-based multiprocessor simulation tool [11] is used in this study to compare the performance of the eager and the deferred coherence protocols using the SPLASH-2 workload [12]. We assume each instruction takes one cycle to execute using a perfect branch predictor when the instruction is found in the L_1 instruction cache. The

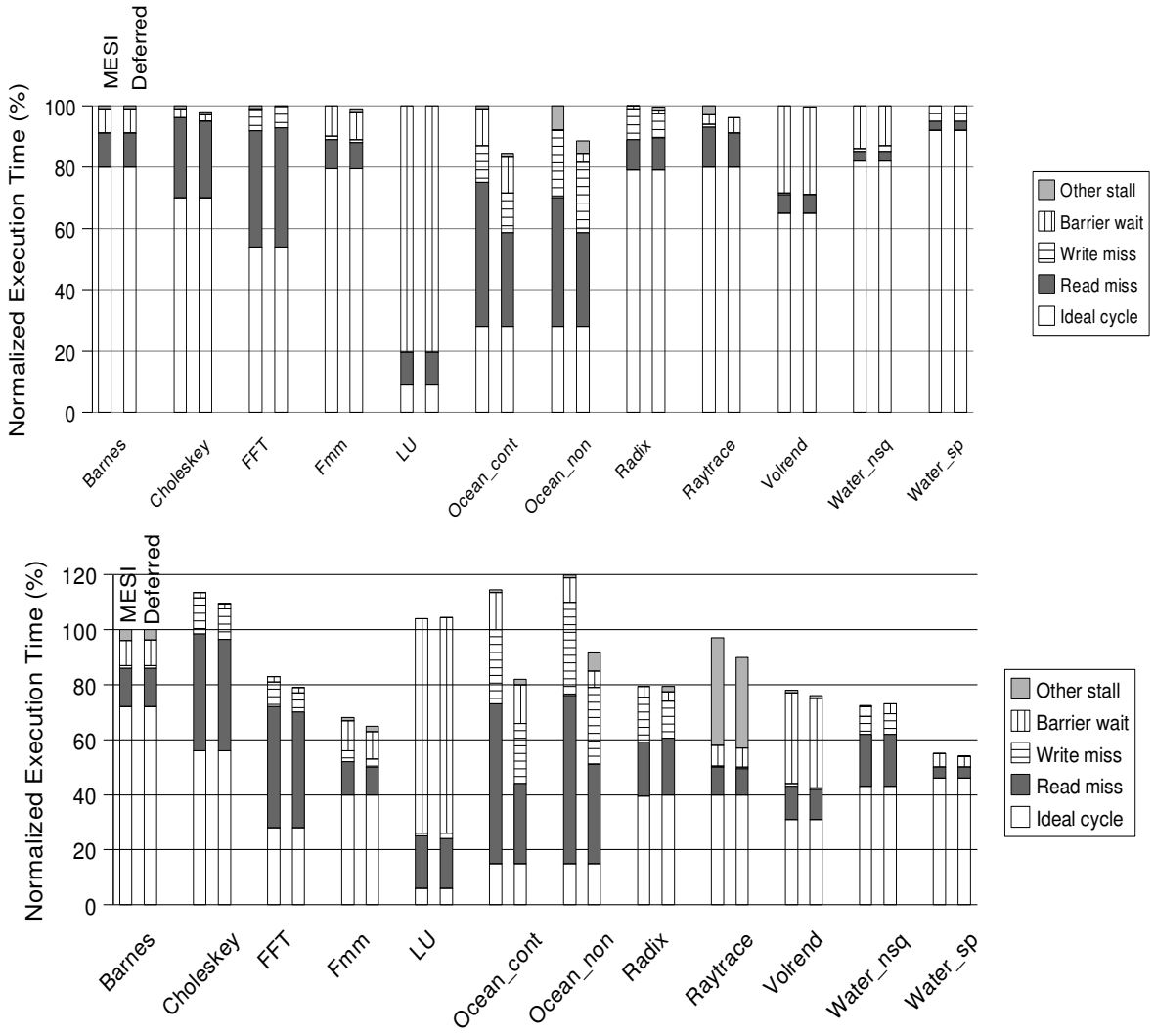

Figure 2: Normalized Execution Time of SPLASH-2 Applications: upper(P=32), lower(P=64)

load/store instruction also takes a single cycle if both the instruction and the data are located in the L_1 caches. A delay of 4 cycles is charged when the instruction or the data is not located in the L_1 cache, but present in the L_2 cache. Further delays are incurred according to detailed bus/memory cycles when the requested instruction and data miss the L_2 cache. We assume a separate direct-mapped instruction and data L_1 cache of 8 Kilo-Bytes (KB) with a 512KB combined 4-way set-associative L_2 cache. Note that we simulate small L_2 caches because of small data sizes in SPLASH-2 applications. The cache line size is 64 bytes for both L_1 and L_2 caches. We further assume a write-miss will not stop the processor pipeline until another miss is encountered or the write buffer is full.

Split-transaction snooping buses based on the MESI and the deferred coherence protocols are modeled. In view of the current technology, we assume the processor cycle is four times faster than the bus cycle. The snooping bus consists of separate *command/address bus* and *data bus*. The width of the data bus is 16 bytes and a 64-byte cache line can be transferred in 4 consecutive cycles. The command bus can accept a request every three cycles. Once a request is active, it requires two cycles for bus arbitration. The command and address

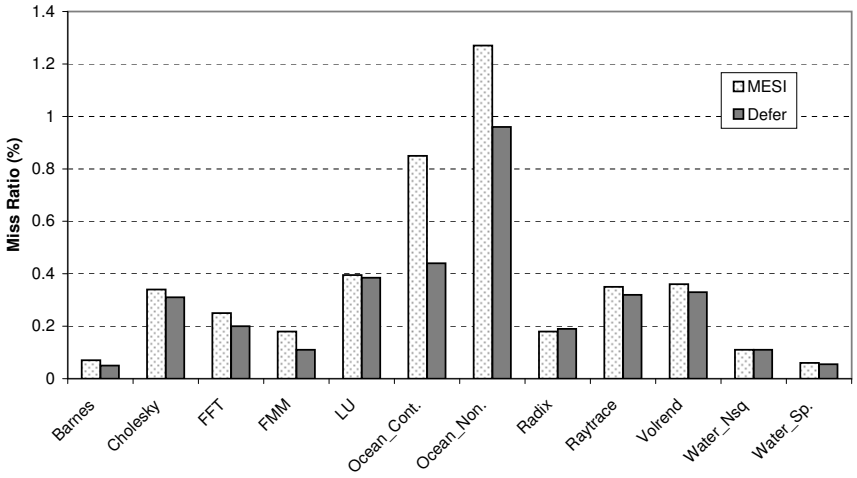

Figure 3: L_2 miss ratio comparison (P=64)

are issued right after the bus is granted. It then takes two cycles for each processor to look-up and update the cache directory for the snooping request. The processor receives the data in seven bus cycles after issuing a request.

Figure 2 shows the normalized execution of the SPLASH-2 benchmarks on SMPs with 32 and 64 processors. The execution time is normalized with respect to the execution time under the MESI-state coherence protocol on a 32-processor system. The total execution time is divided into 5 timing components. The *Ideal cycle* is the time to execute the respective program without any stall, i.e., the CPI is equal to 1. The *Read/Write miss* penalty represents the delay associated with read/write penalties when the requested data is not in the L_1 cache (also includes coherence miss penalties). The *Barrier wait* characterizes the delay introduced by the load imbalance between barriers. Finally, the *Other stall* is the remaining delays due to snooping bus busy, snooping conflicts, merging of the P-state lines, etc.

A few interesting facts can be found from the figure. First, the ideal execution times of the applications are almost reduced to half for most applications when the number of processors increases from 32 to 64. However, the penalty cycles are considerably larger on 64 processors. In fact, in some cases, such as Cholesky, the total execution time increases on the 64-processor systems for both MESI and deferred coherence schemes. The primary reason is due to bus congestion which causes further delays on read/write misses, write-backs, and cache reconciliations on a 64-processor system than that on a 32-processor system.

In terms of relative performance comparison between the MESI the deferred protocols, the deferred coherence scheme, generally speaking, shows better performance than that of the conventional MESI-state protocol especially on 64-processor systems where the bus is more congested. Even on a 32-processor system, noticeable improvement by the deferred protocol can be seen for the workloads Ocean_Cont., Ocean_Non., and Raytrace. On a 64-processor system, the deferred scheme shows about 6%, 4%, 3%, 30%, 30%, 8%, and 3% improvement of the total execution time over the MESI-state protocol for benchmarks Cholesky, FFT, Fmm, Ocean_Cont., Ocean_Non., Raytrace, and Volrend, respectively. Under the deferred coherence protocol, coherence misses caused by data sharing are reduced as shown in Figure 3. As a result, we can see a noticeable reduction in total read/write miss penalties in Figure 2. For some workloads, however, the performance difference between the MESI and the deferred protocols is very minor as we notice that this set of fine-tuning applications have very little false-sharing behavior [12].

6 Conclusion

The deferred cache coherence protocol with a new partially-modified state is designed and evaluated. The unique feature is to allow inconsistent copies of a modified cache line in multiple caches temporarily to circumvent the adverse effect of the false-sharing behavior in parallel programs. In addition, the two-phase write posting and the efficient merging algorithm further reduce the overhead associated with the deferred coherence model. Execution-driven, cycle-by-cycle simulation of snooping-bus multiprocessor models are developed to evaluate the performance of the proposed method. The results based on parallel applications from SPLASH-2 show that the proposed two-phase coherence protocol can indeed improve the overall performance.

Acknowledgement: This work is supported in part by NSF grant EIA-0073473 and by Intel research donations.

References

[1] S. Adva, and M. Hill, "Weak Ordering - A New Definition," *Proc. of 17th Int'l Symp. on Computer Architecture*, May 1990, pp. 2–14.

[2] J. Carter, J. Bennett, and W. Zwaenepoel, "Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems," *ACM Trans. on Computer Systems,* Vol 13(3), Aug. 1995, pp. 205–243.

[3] M. Dubois, J. Wang, L. Barroso, K. Lee, and Y. Chen, "Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs," *Proc. of the 1991 Conf. on Supercomputing*, Nov. 1991 pp. 197-206.

[4] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom, "The Detection and Elimination of Useless Misses in Multiprocessors," *Proc. of 20th Int'l Symp. on Computer Architecture*, June 1993, pp. 88–97.

[5] S. Eggers, and T. Jeremiassen, "Eliminating False Sharing," *Proc. of 1991 Int'l Conf. on Parallel Processing* Aug. 1991, pp. I-377–I-381.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. of 17th Int'l Symp. on Computer Architecture*, May 1990, pp. 15–26.

[7] L. Knotothanassis, M. Scott and R. Bianchini, "Lazy Release Consistency for Hardware-coherent multiprocessor" *Proc. of Supercomputing '95*, 1995, pp. 1705–1736.

[8] P. Keleher, A.L. Cox, and W. Zwaenepoel "Lazy Release Consistency for Software Distributed Shared Memory" *Proc. of 19th Int'l Symp. on Computer Architecture* , May. 1992, pp. 13–21.

[9] J. Larus, B. Richards, and G. Viswanathan, "LCM: Memory System Support for Parallel Language Implementation," *Proc. of 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 208–218.

[10] D. Lenoski, J. Laudon, K, Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz and M.S. Lam "The Standard Dash Multiprocessor," *Computer, 25(3),* March 1992, pp. 63–79.

[11] J.E. Veenstra and R.J. Fowler, "Mint: A Front End for Efficient Simulation of Shared-Memory Multiprocessors" *Proc. of 2nd Int'l Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication System(MASCOTS '94),* Jan. 1994, pp 201–207

[12] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. of 22nd Int'l Symp. on Computer Architecture*, June 1995, pp. 24–36.