University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

May 2008

Improved Sequence-Based Speculation Techniques for Implementing Memory Consistency

Colin Blundell
University of Pennsylvania, blundell@cis.upenn.edu

Milo M.K. Martin
University of Pennsylvania, milom@cis.upenn.edu

Tom Wenisch
University of Michigan, twenisch@eecs.umich.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
Colin Blundell, Milo M.K. Martin, and Tom Wenisch, "Improved Sequence-Based Speculation Techniques for Implementing Memory Consistency", . May 2008.

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-18.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/881
For more information, please contact repository@pobox.upenn.edu.

Improved Sequence-Based Speculation Techniques for Implementing Memory Consistency

Abstract
This work presents BMW, a new design for speculative implementations of memory consistency models in shared-memory multiprocessors. BMW obtains the same performance as prior proposals, but achieves this performance while avoiding several undesirable attributes of prior proposals: non-scalable structures, per-word valid bits in the data cache, modifications to the cache coherence protocol, and global arbitration.

BMW uses a read and write bit per cache block and a standard invalidation-based cache coherence protocol to perform conflict detection while speculating. While speculating, stores to block not in the cache are placed into a coalescing store buffer until those misses return. Stores are written speculatively to the primary cache, and non-speculative state is maintained by cleaning dirty blocks before being written speculatively. Speculative blocks are invalidated on abort and marked as non-speculative on commit. This organization allows for fast, local commits while avoiding a non-scalable store queue.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-08-18.

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/881

Improved Sequence-based Speculation Techniques for Implementing Memory Consistency

Colin Blundell Milo M. K. Martin Tom Wenisch
University of Pennsylvania University of Michigan

UPenn CIS Technical Report TR-CIS-08-18
May 27, 2008

Abstract

This work presents BMW, a new design for speculative implementations of memory consistency models in shared-memory multiprocessors. BMW obtains the same performance as prior proposals, but achieves this performance while avoiding several undesirable attributes of prior proposals: non-scalable structures, per-word valid bits in the data cache, modifications to the cache coherence protocol, and global arbitration.

BMW uses a read and write bit per cache block and a standard invalidation-based cache coherence protocol to perform conflict detection while speculating. While speculating, stores to block not in the cache are placed into a coalescing store buffer until those misses return. Stores are written speculatively to the primary cache, and non-speculative state is maintained by cleaning dirty blocks before being written speculatively. Speculative blocks are invalidated on abort and marked as non-speculative on commit. This organization allows for fast, local commits while avoiding a non-scalable store queue.

1 Introduction

Store miss latency is a significant performance problem in multiprocessors [4]. Uniprocessors can hide store latency via small and simple coalescing store buffers, but such mechanisms can break the correctness of parallel programs if used indiscriminately (*e.g.*, by making a lock release visible to other processors before the update to the shared variable that it guards [9]). Thus, a multitude of *memory consistency models* have arisen that provide different balances between performance and programmability [1]. Sequential Consistency (SC) [9], which disallows any memory reorderings, is the most intuitive model for the programmer [8]. However, straightforward implementations of SC cannot hide store miss latency. On the other hand, relaxed memory models relax some or all ordering constraints. For example, release Consistency (RC) enforce ordering only at programmer-specified *barriers*. Relaxed models allow the use of store buffers to tolerate store latency, but require difficult reasoning on the part of the programmer to manually identify and annotate code where precise ordering is required for correctness.

There is a long lineage of proposals that speculatively relax ordering constraints at the level of individual loads and stores [5, 6, 12] in order to increase performance. Industry, however, continues to implement relaxed models, although work demonstrating that a SC implementation can match the performance of a straightforward implementation of RC is now nearly ten years old [6]. One likely reason for this fact is that the depth of speculation that these proposals require to fully hide store latency results in structures that are infeasibly large, complicated and/or unscalable. To make SC appealing to industry, designs must not only match or exceed the performance of weak models but must do so without adding significant implementation complexity.

Several recent proposals [3, 7, 15] present a promising initial step in this direction. These proposals implement speculation at the granularity of sequences of instructions rather than that of individual instructions; this advance reduces the overheads required to support deep speculation. Unfortunately, these proposals make significant changes to the cache coherence protocol [3, 7] or add structures whose size must scale linearly with memory latency [15]. Furthermore, the process of committing a speculative sequence can have significant latency in these proposals, requiring support for multiple active speculative sequences to avoid performance degradation.

In this work we present BMW, a new design for speculative consistency that simplifies the proposal by Wenisch et al. [15] (henceforth referred to as ASOsc) without sacrificing performance. The complexity of ASOsc–structure to hold all speculative stores in-order whose size scales linearly with memory latency and per-word valid bits on L1 cache blocks–stems from the fact that its speculation commit process involves draining all stores in-order into the L2 cache (discussed in Section 2). Our key simplifying observation is that it is unnecessary to maintain the order of stores within a speculative sequence (beyond dataflow order), because all the stores in the sequence become visible atomically.

Like ASOsc, BMW initiates speculation whenever the processor would otherwise stall due to the constraints of SC. However, BMW employs a coalescing store buffer like those used in uniprocessors rather than the in-order, unscalable structure of ASOsc. During speculation a processor is free to use the store buffer to tolerate store latency exactly as in a uniprocessor. Processors detect violations of atomicity by snooping cache coherence requests on speculatively-accessed cache blocks as in ASOsc and other proposals. When all store misses in a speculative sequence are satisfied, a processor commits speculation in constant time by

flash-clearing the speculatively-accessed bits on L1 cache blocks.

BMW has several features that distinguish it from prior proposals:

Minimal hardware changes. The only features that BMW requires over a high-performance RC implementation are the ability to take a register checkpoint and 2 bits per cache block.

Constant-time speculation commit. Because of its constant-time commit, BMW with support for one speculative sequence at a time achieves as high performance as that of ASOsc with support for four speculative sequences at a time.

Synergy with speculative synchronization. Speculative synchronization proposals using eager conflict detection [10] can support BMW at little added cost, analogous to the findings of TCC [7] and BulkSC [3] in the context of lazy conflict detection.

2 Background

In this section we first provide a brief overview of memory consistency definitions and traditional implementations and then detail ASOsc [15], a recent proposal for implementing sequential consistency via checkpoint/recovery-based processing [2].

2.1 Memory Consistency Models: Definitions and Traditional Implementations

Tolerating store miss latency in uniprocessors. In a uniprocessor, a store miss that reaches the head of the instruction window is retired and moved to a store buffer, which is sourced on loads. This store buffer is free to coalesce stores to the same address and complete stores to the cache in any order as requests are filled.

The store miss latency problem in multiprocessors. As discussed in Section 1, all multiprocessors must support a memory consistency model as part of their software interface. There are three general varieties of memory consistency models: those that strictly enforce memory order (typified by Sequential Consistency or SC), those that relax store-to-load order but enforce store-to-store order (typified by Processor Consistency or PC), and those that relax all order except at programmer-specified ordering points called *barriers* (typified by Release Consistency or RC).

All three models restrict a multiprocessor's ability to use a store buffer (and thus, to tolerate store miss latency) to different degrees. Under SC a load cannot retire until all previous stores complete, rendering a store buffer essentially useless. PC relaxes this restriction but still requires stores to complete in-order,

causing backpressure at instruction retirement due to the store buffer filling. RC implementations can use a coalescing store buffer but must drain the store buffer before proceeding past a barrier; previous work has shown that even this requirement can cause store miss latency to form a significant component of total execution time [14, 13].

Speculative consistency model implementations. Many proposals have sought to increase the performance of SC implementations through speculation, using techniques such as speculative load execution and non-binding store prefetching [5], speculative load retirement [12], and speculative store completion [6]. These optimizations improve the performance of SC implementations significantly. However, they suffer from the fact that as memory latency grows, the depth of speculation needed to hide the latency of a store miss also grows. In-window mechanisms such as speculative load execution are thus not able to fully hide store latency, while mechanisms that allow instructions to speculatively leave the instruction window (such as the SHiQ [6]) would have to scale to sizes that are impractical to support the depth of speculation required. Thus, these proposals do not fully solve the store latency problem in multiprocessors supporting SC.

2.2 ASOsc: A Checkpoint/Recovery-Based Speculative SC Design

Recently Wenisch et al. have proposed ASOsc [15], a new speculative SC design based on the idea of checkpoint/recovery-based processing [2]. Below, we first describe the concept of checkpoint/recovery processing, and then describe how Wenisch et al. implement this concept in the context of speculative SC.

Checkpoint/recovery processing. Checkpoint/recovery processing is a technique to support deep speculation. The difference between checkpoint-based speculation and traditional in-window speculation (*e.g.*, on a branch prediction), is that the former performs speculation at a coarse granularity rather than on a per-instruction basis. To initiate speculation the processor takes a register checkpoint. During speculative execution the processor ensures that all memory accesses occur atomically, that is that no other processor accesses the same memory in a conflicting way. To commit the speculative sequence the processor makes speculative memory state globally visible, again atomically.

An implementation of this technique must resolve two important questions: first, how to detect violations of atomicity, and second, how to maintain memory state such that speculative stores can be made visible atomically on commit and their effects undone on abort.

Wensch et al. observed that checkpoint/recovery processing is a promising candidate to simplify speculative SC implementations: the processor maintains ordering at the level of atomic sequences of instructions rather than at the level of individual instructions. ASOsc initiates a speculative sequence whenever consistency constraints would otherwise force the processor to stall. Below we detail how their design supports the mechanisms required for checkpoint/recovery-based speculation. We then detail the features and shortcomings of this design.

Violation detection. During speculative execution they detect violations of atomicity by having speculative accesses set "speculatively read" and "speculatively written" bits on cache blocks. External cache coherence requests snoop these bits and trigger an abort if they detect a conflict (*i.e.*, a write request for a speculatively-read block or any request for a speculatively-written block).

Maintenance of speculative memory state. To support speculative memory state ASOsc replaces a traditional store buffer with a structure called the Scalable Store Buffer (SSB). Each speculative store writes its address and value in-order into this buffer. To avoid the requirement of the SSB having to source loads, all stores also write their value into the L1 cache (whether or not the block is currently present in the L1 cache), and the L1 cache sources data for all loads. On a commit, the buffer is drained in-order into the globally-visible L2 cache. On an abort the buffer is cleared and speculatively-written L1 cache blocks are invalidated.

To tolerate the latency of the speculation commit process ASOsc supports multiple active speculative sequences at a time. L1 cache blocks are extended to support multiple speculatively read/written bits, and ordering points are inserted into the SSB to distinguish the speculative stores of one sequence from that of another. This also enables finer-grained abort: after aborting a speculative sequence ASOsc replays all older speculative stores into the L1 cache, enabling the processor to restart from the beginning of the aborted sequence rather than having to restart from the beginning of the oldest active sequence.

Discussion. Via these mechanisms ASOsc equalizes the performance of SC with that of RC. However, the proposal has some drawbacks. The most significant drawback is that the SSB is a non-coalescing structure. This fact implies that it must be large enough to hold all stores that occur during the duration of a store miss, *i.e.* the size of the SSB must scale with memory latency. The commit process of ASO is also linear in the number of stores that occur in a speculative sequence. Thus, the number of active

sequences required to hide this latency would grow with memory latency as well. A smaller issue is the SSB's requirement of per-byte valid bits on cache blocks.

3 BMW

In this section we propose a design for speculative consistency that remedies the flaws of ASOsc. Our main goals are first, to eliminate any structures whose size must scale linearly with memory latency, and second, to have a local speculation commit process whose duration is independent of the size of the speculative sequence being committed. To achieve these goals, we focus on removing the Scalable Store Buffer (SSB). While the SSB's total store order ensures that stores can commit properly to the L2 cache on commit of a speculative sequence and allows finer granularity of rollback on abort, it is also the source of the flaws of ASOsc: the requirement of total order disallows coalescing of stores, resulting in the SSB's non-scalability and a commit process whose duration is linear in the number of stores in the speculative sequence. Aside from the changes that we describe below, our design is identical to ASOsc.

To replace the SSB, we exploit the observation that within a speculative sequence relative store order does not need to be maintained (beyond dataflow) because the stores will all commit atomically at the end of the speculative sequence (an observation previously made in other contexts [11]). Thus, instead of the SSB, we combine ASOsc's mechanism for violation detection with a traditional coalescing store buffer for speculative memory state. Below we describe how this works.

Violation detection. Identical to ASOsc.

Speculative memory state. BMW employs a traditional coalescing store buffer that sources loads. During speculative execution, the store buffer operates exactly as in a uniprocessor, holding data for store misses until they complete. The processor can commit speculation whenever the store buffer is empty simply by clearing the speculatively read/written bits. To ensure that the processor can roll back in the event of an abort, speculative stores that write to a dirty cache block whose speculatively-written bit is not set first write back the old value to the L2 cache, thus preserving that old value in the case of an abort. On such an abort the processor invalidates any speculatively-written blocks; these blocks will be refetched from the L2 on demand.

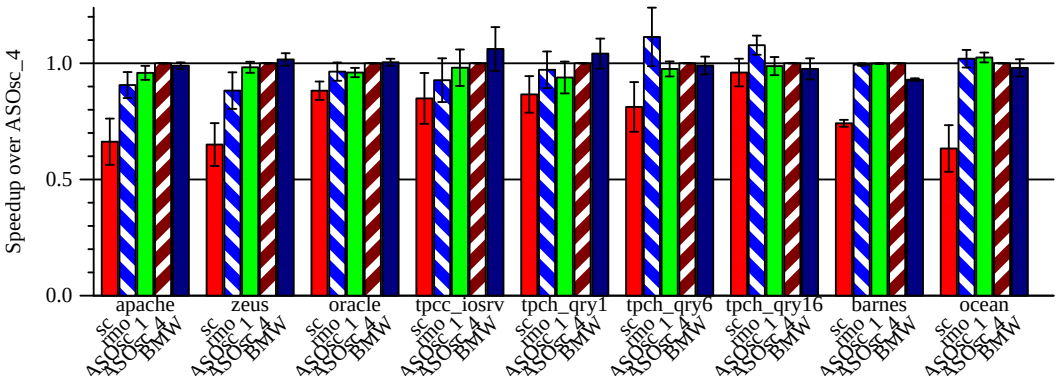

Figure 1: Performance of BMW versus ASOsc.

Discussion. BMW has several advantages over ASOsc: rather than the SSB (whose size grows with the number of stores in a speculative sequence), we use a conventional, well-understood coalescing unordered store buffer; we do not need per-byte valid bits on L1 cache blocks; our commit-time process is constant-time rather than linear in the size (number of stores) of the sequence being committed. However, BMW is also has some potential drawbacks relative to ASOsc. As described, our design does not support multiple active sequences at a time. Because of this, it can neither hide speculation commit latency nor perform fine-grained abort. In the next section we will show that in spite of these facts BMW performs as well as ASOsc on commercial workloads and will analyze the reasons why this holds.

4 Evaluation

This section evaluates our proposal.

4.1 Experimental Methodology

We use the full-system simulator Flexus [16]. We compare the performance of our design to ASOsc on the workloads used in the Store-Wait-Free Multiprocessors paper [15]. Our goal is to determine whether the simplifications that our design makes to ASOsc–non-pipelined commit and coarser-grained abort–negatively impact performance.

4.2 Presentation and Analysis of Results

Our experimental results (shown in Figure 1) show that the performance of BMW is competitive with that of ASOsc in all cases. The reasons that the performance of BMW is not hurt by its simplifications of ASOsc are that (1) most sequences commit opportunistically when the processor sees that the store buffer is

empty (eliminating the need for pipeline commit), and (2) abort is extremely rare (rendering irrelevant the finer-granularity abort of ASOsc). .

5 Conclusions

In this work we proposed BMW, a novel speculative SC implementation that adds negligible complexity (only 2 bits per L1 cache block and the ability to take a register checkpoint) to a traditional high-performance RC implementation. BMW builds on the previously-proposed design of ASOsc, adopting its checkpoint/recovery-based operation and use of the cache coherence protocol for violation detection but replacing its large and unscalable SSB with a traditional coalescing store buffer. We quantitatively evaluated BMW against ASOsc and showed that it performed as well in all cases, with the primary reason being that the ability of BMW to opportunistically commit in constant-time when the store buffer is empty means that the lack of support for multiple active sequences does not harm performance.

The simplicity of our design has important implications for multiprocessor manufacturers. First, our design is simple enough to be incorporated into multiprocessors supporting PC/TSO and even RC/RMO (for speculation past barriers), whereas previous designs had arguably been too complex to be appealing to designers supporting consistency models whose main appealing feature is implementation simplicity. More importantly, our design is synergistic with previous proposals for program-level speculative synchronization. Designers who add conflict detection bits to cache blocks for transactional memory could reuse those bits to implement speculative memory consistency. Interestingly, the functionality required by our system is a subset of that required for transactional memory, which must support unbounded transactions whereas we need to support only bounded speculative sequences. Thus, designers who implement transactional memory using speculatively read and written bits on cache blocks can get a high-performance consistency model implementation virtually for free.

References

[1] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.

[3] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[4] Y. Chou, L. Spracklen, and S. G. Abraham. Store Memory-Level Parallelism Optimizations for Commercial Applications. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 183–196, Nov. 2005.

[5] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, volume I, pages 355–364, Aug. 1991.

[6] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, May 1999.

[7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 102–113, June 2004.

[8] M. D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, 31(8):28–34, Aug. 1998.

[9] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[10] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th Symposium on High-Performance Computer Architecture*, Feb. 2006.

[11] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2001.

[12] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, June 1997.

[13] O. Trachsel, C. von Praun, and T. Gross. On the Effectiveness of Speculative and Selective Memory Fences. In *Proceedings of the International Parallel and Distributed Processing Symposium Symposium*, Apr. 2006.

[14] C. von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu. Conditional Memory Ordering. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 41–52, June 2006.

[15] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[16] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26(4):18–31, 2006.