Appears in the IEEE Micro: Micro's Top Picks from Computer Architecture Conferences, Nov/Dec 2004

Coherence Decoupling: Making Use of Incoherence

Jaehyuk Huh [§] Jichuan Chang[†] Doug Burger [§] Gurindar S. Sohi[†]

[§]Department of Computer Sciences
The University of Texas at Austin

[†]Computer Sciences Department
University of Wisconsin-Madison

Abstract

Coherence decoupling uses speculation to reduce the effects of coherence miss latencies. This approach breaks conventional cache coherence protocols into two separate types of protocols: one for the speculative use of data (performance) and one for the eventual verification of coherent shared data (correctness). By using values in a local cache's invalid lines, the system can hide false sharing latencies, while by speculatively writing updates to remote invalid lines, the overheads of true sharing can be reduced.

1 Introduction

Multiprocessing and multithreading are becoming ubiquitous even on single chips. With increasing cache sizes, coherence misses in such systems will account for a larger fraction of all cache misses. As communication latencies increase, the increased fraction of coherence misses will cause significant and increased performance losses. Reductions in communication latencies can be achieved by tuning coherence protocols for specific communication patterns and/or applications. However, these optimizations increase the design complexity of the protocol, which makes them difficult to verify. A competing approach requires parallel programmers to tune applications to work well with simpler protocols — for example, padding data structures to reduce false sharing, at the cost of decreased programmer productivity.

Computer architects have successfully applied speculative execution to improve performance in a variety of scenarios. In this paper, we propose a new type of load speculation called *coherence decoupling*, which

uses speculation to reduce the effect of long communication latencies, without exacerbating the programmer's task or complicating the coherence protocol. Coherence decoupling breaks the communication of a value into two constituent parts: (i) the acquisition and use of the value, and (ii) the communication of coherence permissions that indicate the correctness of the value and validate its usage. In traditional cache coherence systems, these two aspects are coupled within a single protocol. This coupling requires strict acquisition of the coherence permissions before the use of data, thus serializing the two. Coherence decoupling implements separate protocols for the speculative use and eventual verification of values. A *Speculative Cache Lookup* (SCL) protocol provides speculative values as quickly as possible, while in parallel a simple, backing *coherence protocol* progresses and eventually produces the correct values along with their requisite permissions.

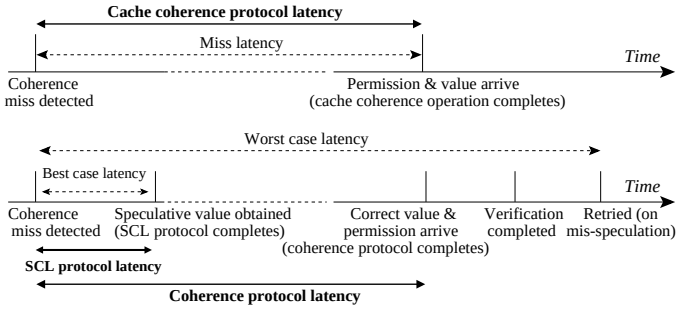

Figure 1: Coherence Decoupling

Separating the SCL protocol and the coherence protocol allows each to be tuned independently, accelerating communication with less complexity than conventional protocol optimizations. The separation also allows the overlap of the two protocols. In Figure 1, compared to a conventional coherence protocol, coherence decoupling uses the SCL protocol to return the speculative data early. This early return can occur if a request matches the tag in the local cache, as the processor simultaneously launches the invalid-to-shared request via the coherence protocol. When the coherence protocol returns the correct value and its permissions, the returned value is compared to the speculatively used value, which is buffered in the MSHR. If they are identical, the speculation was correct, and the coherence latency will have been partially or fully overlapped with useful computation ("best case" in the figure). If the values differ, a rollback must occur, resulting in a performance loss compared to no speculation ("worst case" in the figure).

The utility of coherence decoupling, as with all speculation policies, depends on the ratio of correct to incorrect speculations, the benefits of successful speculations, and the cost of recovery.

We evaluate several different SCL protocols with varying speculation accuracies, while maintaining a simple, invalidation-based coherence protocol for correctness. The basic SCL protocol merely accesses the data in the local cache if the tag matches, which greatly reduces performance losses due to false sharing. The next set of SCL protocols are variants of a *write-update* protocol. These protocols trades off speculation accuracy with extra traffic used to distribute speculative writes. Unlike canonical write-update protocols, which suffer from design complexity, write-update SCL protocols only change data in invalid lines, making them simple and easy to verify.

2 Accelerating Coherent Accesses

Although coherence decoupling is a new approach, much previous work targeted similar goals. The most relevant prior work falls into three broad categories: (1) customized coherence protocols that reduce communication latency by adapting to specific sharing patterns and/or applications [4]; (2) speculative coherence operations that predict coherence operations (not operation results) and initiate speculative invalidations or upgrades accordingly [3]; and (3) speculation on the outcome of events in a multiprocessor execution, including speculative synchronization [9] and speculating on the execution's conformance to a strong memory model [1]. Coherence decoupling differs from prior work in that it both speculates on coherence results (data values) and allows for decoupling performance and correctness protocols (similar to Token Coherence [8]). This technique is essentially a form of load value prediction (LVP) [6], but one which uses a different mechanism to obtain speculative values (using invalid cached values rather than values from a speculation table or state machine), which also permits the separation of the coherence protocol into two the classes of protocols. Coherence decoupling speculates correctly either when data are falsely shared, or when updated by either a silent store, a temporally silent store [5], or a speculative write update.

3 Coherence Decoupling Architecture

Coherence decoupling separates a cache coherence protocol into two parts: (i) a speculative cache lookup (SCL) protocol, which produces a speculative

SCL Component	Policy	Description
Read	`CD`	Use the locally cached value
Read	`CD-F`	Add a PC-indexed confidence predictor to filter speculations
Update	`CD-IA`	Use invalidation piggyback to update all invalid blocks
Update	`CD-N`	Update sharers after N writes to a block (N=5 in evaluation)

Table 1: SCL Protocol Components

value to be used for further computation, and (ii) a coherence protocol, which returns the correct value (as defined by the memory consistency model) and the permissions to use the value. When the SCL protocol returns a value sooner than the coherence protocol, the computation using the value can be overlapped with the coherence operation. Accurate SCL protocols hide coherence latencies, allowing simpler but lower performance coherence protocols to be used without a commensurate performance penalty.

To support coherence decoupling, the system architecture must: (i) *split*, breaking a memory operation into a speculative read and a coherence operation, (ii) *compute*, providing mechanisms to execute with the speculative value, and (iii) *recover*, supporting mis-speculation detection and recovery.

Splitting a memory operation (*i* above) into two sub-operations is straightforward. To support speculative computation (*ii* above), the same mechanisms to support other forms of speculative execution can be used, although the growing coherence latencies require mechanisms that can buffer speculative state across hundreds to thousands of instructions. The recovery mechanism (*iii* above) buffers the speculative value in an MSHR, compares it against the value returned by coherence protocol, and recovers if a mis-speculation occurs.

3.1 Correctness of Coherence Decoupling

Using a speculative value from an SCL protocol — and later verifying the speculation via the coherence protocol — is analogous to carrying out a memory operation speculatively assuming that the memory consistency model will not be violated. As Martin et al. have observed [7], implementing value speculation correctly requires the same hardware as that used for aggressive implementation of sequential consistency (SC). With this hardware support, coherence decoupling can be correctly implemented without violating the memory consistency model.

3.2 SCL Protocols for Coherence Decoupling

Backed by a simple and easily verifiable coherence protocol, many different SCL protocols can be implemented to improve performance. Each SCL protocol has a *read* component and a *update* component (which may be null). The read component obtains speculative values and the update component speculatively sends writes to invalid cache lines (possible sharers) to improve the accuracy of future speculations. As long as the system can recover from an invalid value that changed, it is always safe to speculatively write into lines so long as they are also in an invalid state. Table 1 summarizes the small number of SCL protocols that we evaluate in this paper. Many other protocols are possible, some of which are evaluated elsewhere [2].

3.2.1 SCL Protocol Read Component

The first read component policy simply returns the value in the local cache if the block is present (i.e., the tag matches), even if the coherence state is invalid. We call this `CD`, for basic coherence decoupling. This protocol speculates correctly if the accessed word is falsely shared, or updated by a silent store or temporally silent stores, thus simplifying the programmer's task of code tuning to reduce false sharing.

The next read policy (called `CD-F`) adds a PC-indexed confidence predictor to throttle low-confidence speculations. It reduces the number of times speculation is employed (i.e., coverage), but improves the average speculation accuracy over the `CD` protocol.

In general, the read component of an SCL protocol could return a value from an invalid (or valid) line anywhere in the system. Its usefulness depends on the read latency and accuracy. In a directory-based system, for example, the SCL protocol could first access the local invalid line and then the home memory, or even a geographically-proximate remote cache in a hierarchical multiprocessor built from CMPs. In this work, we consider flat, snooping SMPs only, since our simulation infrastructure is not able to evaluate large-scale or hierarchical systems.

3.2.2 SCL Protocol Update Component

The *update* component is added to an SCL protocol to improve the speculation accuracy for truly shared data, by writing updates to invalid lines around the system. The extra bandwidth consumed is traded for increased speculation accuracy.

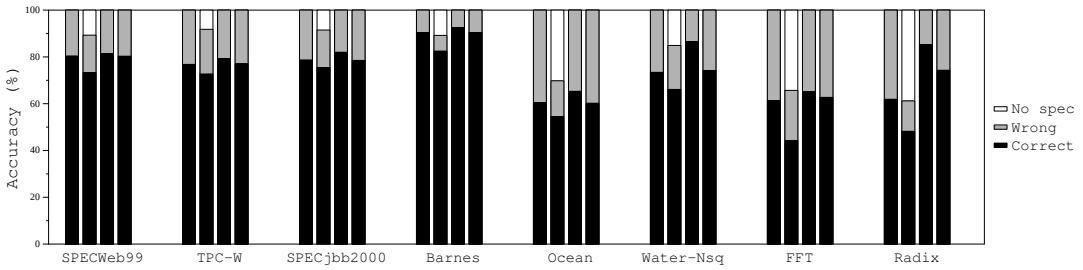

Figure 2: Accuracy of Coherence Decoupling (from left to right: `CD`, `CD-F`, `CD-IA`, `CD-N`)

Our first update policy, `CD-IA`, piggybacks the write value along with the writer's invalidation message. In a bus-based system, `CD-IA` updates the value in all caches that have the block in invalid state, not only those transitioning from a shared to an invalid state. The other policy, `CD-N`, broadcasts the dirty line after N writes have been made by the writer. Additional messages are required for such updates.

Both protocols are variants of a write-update protocol, offering different accuracies and bandwidth requirements. They are different from a canonical write-update coherence protocol because the speculative updates are completely non-blocking for the writer and also because the updates could be dropped at any point in the system without affecting correctness.

4 Performance Evaluation

We ran our experiments on *MP-Sauce*, an execution-driven, full-system multiprocessor timing simulator, limiting the simulations to 16-node SMP systems. We simulated three commercial applications and five scientific shared-memory benchmarks from the SPLASH2 suite.

4.1 Coherence Decoupling Accuracy

Figure 2 shows the ratio of correct to incorrect CD speculations (for all coherence misses) using a 4MB cache with 128-byte blocks, for the policies described in Table 1. In the `CD-N` experiment, we updated the invalid sharers after the first 5 writes to a line.

The `CD-F` policy is the only one to not speculate on all coherence misses, due to its filter which blocks low-confidence speculations. The base `CD` protocol makes more correct speculations than `CD-F`, but at the expense

Benchmark	CD	CD-F	CD-IA	CD-N5	Optimal
SPECWeb99	13.8%	11.0%	13.2%	14.9%	34.6%
TPC-W	1.2%	2.6%	2.3%	1.4%	17.8%
SPECjbb2000	16.6%	15.8%	13.5%	17.1%	26.3%
Barnes	0.6%	0.4%	0.7%	0.8%	1.4%
Ocean	6.9%	4.7%	8.2%	6.0%	34.5%
Water-Nsq	2.1%	1.7%	2.8%	0.7%	17.4%
FFT	5.1%	4.2%	6.1%	4.6%	21.4%
Radix	6.8%	3.6%	7.6%	6.3%	42.4%
Mean	6.6%	5.5%	6.8%	6.5%	24.5%

Table 2: Speedups for Coherence Decoupling

of more mispredictions. However, this simple protocol provides accuracies that approach those of many of the update protocols, due to silent stores and false sharing. For three commercial benchmarks and Barnes, the base `CD` protocol can predict correct values for more than 70% of coherence misses. Some of the update protocols lose accuracy by sending the update too early, changing an invalid line to a new value, after which the writer changes the value *back* (a temporally silent store) but may not broadcast the change, resulting in a mis-speculation. Overall, coherence decoupling appears to have much better accuracies for the commercial workloads, with the simplest `CD` protocol performing as well as the more complex protocols, except on a few of the simpler scientific codes.

4.2 Coherence Decoupling Timing Results

Table 2 shows the speedups over the baseline system (which is the simple invalidation protocol with no coherence decoupling or speculation). We model a flushing mechanism to recover from mis-speculations. The mechanism flushes all instructions younger in program order when the violation is detected (a "rolling flush") rather than waiting until the violation reaches the head of the reorder buffer.

The right-most column of Table 2 places an upper bound on the performance of coherence decoupling in the simulated system. In this model, all cache accesses that would have been coherence load misses are treated as hits. SPECWeb99 and Ocean show large ideal benefits (34.6% and 34.5%), but Barnes shows a mere 1.4% due to its negligible L2 miss rates.

The accuracies of coherence decoupling are high, partially or fully tolerating a third to a half of coherence misses. The speedups reflect those results for several benchmarks; in particular, SPECJbb reaches over half of

its ideal performance improvement for most of the policies. Overall, with only simple mechanisms, the base CD policy achieves a mean speedup of 6.6%, which is over a quarter of the ideal speedup. In larger-scale systems (and particularly CC-NUMA systems), the speedups will likely be much higher. In those systems, remote coherence latencies–especially those that take multiple hops across the network–will have a more deleterious effect on performance.

5 Conclusions

Coherence decoupling is a microarchitectural mechanism that reduces the cost of coherence communication, mitigating the burden on both the coherence protocol designer and, more important, the parallel programmer. An early return of a speculative value allows further useful computation to proceed in parallel with the coherence correctness protocol, overlapping long coherence latencies with useful computation. Furthermore, decoupling the SCL protocol (which returns a speculative value quickly) from the coherence protocol (which ensures the correctness of the value) allows each protocol to be optimized separately. The SCL protocol can be optimized for performance since it does not have to ensure correctness; the coherence protocol can be simple since its performance is not paramount. Further work in this area involves improving SCL protocols to further increase speculation accuracies, and recovering from mis-speculations more efficiently. Another open direction is the utility of coherence decoupling for both hierarchical multiprocessors and multiprocessors with directory-based cache coherence.

6 Acknowledgments

This material is based on work supported in part by the Defense Advanced Research Project Agency (DARPA) under Contracts NBCH30390004 and F33615-03-C-4106, the National Science Foundation under grants EIA-0071924, CCR-0311572, and CCR-9985109, support from the Intel Research Council, an IBM Faculty Partnership Award, and the University of Wisconsin Graduate School.

7 Biography

- Jaehyuk Huh is a PhD student in CS at UT-Austin. His research interests include microarchitecture, multiprocessors and operating systems.

- Jichuan Chang is a PhD student in computer sciences at University of Wisconsin-Madison. His research interests include multiprocessor cache coherence and speculation in chip-multiprocessors.

- Doug Burger is an Associate Professor of Computer Sciences at UT-Austin. Dr. Burger received a PhD in computer science from the University of Wisconsin in 1998. His current research interests focus on high-performance, power-efficient, technology-scalable microprocessors, and he co-leads the TRIPS project at UT-Austin, which is addressing those challenges. He is also an Alfred P. Sloan Foundation fellow, a senior member of the IEEE and a member of the ACM.

- Guri Sohi received a Ph.D from the University of Illinois in 1985 and has been a faculty member at the University of Wisconsin-Madison since graduation. He is currently the Chair of the Computer Sciences Department. Sohi's research has been in the design of high-performance computer systems. He received the 1999 ACM SIGARCH Maurice Wilkes award "for seminal contributions in the areas of high issue rate processors and instruction level parallelism". At the University of Wisconsin he was selected as a Vilas Associate in 1997 and won the WARF Kellett Mid-Career Faculty Researcher award in 2000. He is a Fellow of the ACM and the IEEE.

Direct questions and comments about this article to Doug Burger, Department of Computer Sciences, The University of Texas at Austin, 1 University Station C0500, Austin, TX, 78712; dburger@cs.utexas.edu.

References

[1] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory. In *Proceedings of the 17th Int. Symp. on Computer Architecture*, May 1990.

[2] J. Huh et al. Coherence decoupling: Making use of incoherence. In *Proceedings of the 11th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[3] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Int. Symp. on Computer Architecture*, June 1995.

[4] D. Lenoski et al. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3), Mar. 1992.

[5] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *Proceedings of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[6] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Architectural Support for Programming Languages and Operating Systems*, 1996.

[7] M. M. K. Martin et al. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proceedings of the 34th Int. Symp. on Microarchitecture*, Dec. 2001.

[8] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Int. Symp. on Computer Architecture*, June 2003.

[9] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Int. Symp. on Microarchitecture*, Dec. 2001.