SPECULATIVE INCOHERENT CACHE PROTOCOLS

COHERENCE DECOUPLING IS A MICROARCHITECTURAL MECHANISM THAT IMPLEMENTS SEPARATE PROTOCOLS FOR SPECULATIVE USE AND FOR THE EVENTUAL VERIFICATION OF VALUES. THE TECHNIQUE REDUCES THE EFFECT OF LONG COMMUNICATION LATENCIES WHILE MITIGATING THE BURDENS ON THE COHERENCE PROTOCOL DESIGNER AND THE PARALLEL PROGRAMMER.

Jaehyuk Huh
Doug Burger
The University of Texas at Austin

Jichuan Chang
Gurindar S. Sohi
University of Wisconsin-Madison

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, this larger fraction of coherence misses will cause significant and increased performance losses. Tuning coherence protocols for specific communication patterns and applications can reduce communication latencies. However, these optimizations increase a protocol's design complexity, making the protocol difficult to verify. A competing approach requires parallel programmers to tune applications to work well with simpler protocols—for example, by padding data structures to reduce false sharing—at the cost of decreased programmer productivity.

Speculative execution has successfully improved performance in various scenarios. We propose a new type of load speculation, called coherence decoupling. This technique uses speculation to reduce the effect of long communication latencies without exacerbating the programmer's task or complicating the coherence protocol. Coherence decoupling breaks value communication into two constituent parts: the acquisition and speculative use of the value, and the communication of coherence permissions that indicate the value's correctness and validate its use. 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 aspects. Coherence decoupling implements separate protocols for speculative use and for the eventual verification of values. A speculative cache lookup (SCL) protocol provides speculative values as quickly as possible for use in further computation while a simple backing coherence protocol operates in parallel and eventually produces the correct values (as defined by the memory consistency model), along with their requisite permissions.

Separating the SCL protocol and the coherence protocol enables designers to tune and optimize them independently, accelerating communication with less complexity than with conventional protocol optimizations. Separation also lets the two protocols overlap. Figure 1 shows that, unlike a conventional coherence protocol, coherence decoupling uses the SCL protocol to return the speculative data early. This early return occurs if a request matches the tag in the local cache, at which point the processor simultaneously launches the invalid-to-shared request via the coherence protocol. When the coherence protocol returns the correct value and its

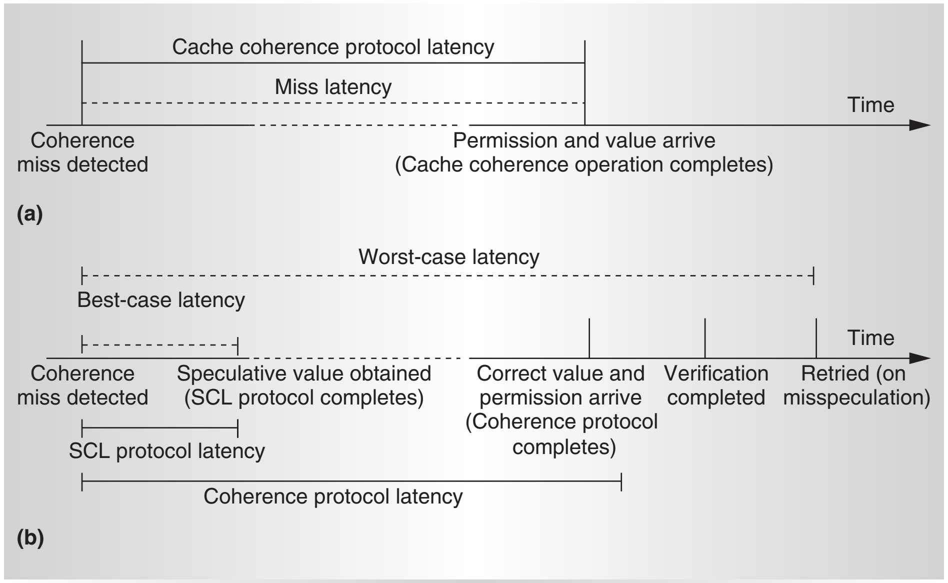
Cache coherence protocol latency
Miss latency
Time
Coherence miss detected
Permission and value arrive (Cache coherence operation completes)
(a)
Worst-case latency
Best-case latency
Time
Coherence miss detected
Speculative value obtained (SCL protocol completes)
Correct value and permission arrive (Coherence protocol completes)
Verification completed
Retried (on misspeculation)
SCL protocol latency
Coherence protocol latency
(b)

Figure 1. A conventional cache coherence protocol (a); coherence decoupling (b).

permissions, the cache controller compares the returned value with the speculatively used value, which is buffered in the miss status holding register (MSHR). If they match, the speculation was correct, and the coherence latency will have been partially or fully overlapped with useful computation (shown in Figure 1b as best-case latency). Nonmatching values, however, require a rollback, resulting in a performance loss that is as great or greater than the loss resulting from no speculation (worst-case latency in Figure 1b). 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 evaluated several SCL protocols with varied 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 resulting from false sharing. A second category of SCL protocols are variants of a write-update protocol. These protocols trade off increased speculation accuracy for increased interprocessor traffic, used to distribute speculative writes. Unlike canonical write-update protocols, which suffer from design complexity, write-update SCL protocols change data only in invalid lines, making the protocols simple and easy to verify.

Accelerating coherent accesses

Although coherence decoupling is a new approach, much previous work had similar goals. The most relevant prior work falls into three broad categories:

- customized coherence protocols that reduce communication latency by adapting to specific sharing patterns and applications;[1]
- speculative coherence operations that predict coherence operations (not operation results) and initiate speculative invalidations or upgrades accordingly;[2] and
- speculation on the outcome of events in a multiprocessor execution, including speculative synchronization[3] and speculation on the execution's conformance to a strong memory model.[4]

Coherence decoupling differs from prior work in that it both speculates on coherence results (data values) and supports decoupling performance and correctness protocols (similar to token coherence[5]). The technique is essentially a form of load value prediction,[6] but one that uses a different mechanism to obtain speculative values. (It uses invalid cached values rather than values from a speculation table or state machine.) Using invalid cached values permits separation of the

Table 1. SCL protocol components.

SCL component	Coherence decoupling policy	Description
Read	Basic (CD)	Use the locally cached value
Read	Filtered (CD-F)	Add a PC-indexed confidence predictor to filter speculations
Update	Invalidation-all update (CD-IA)	Use invalidation piggyback to update all invalid blocks
Update	With N writes (CD-N)	Update sharers after N writes to a block (N = 5 in evaluation)

coherence protocol into the two protocol classes. Coherence decoupling speculates correctly when data is falsely shared or when it is updated by a silent store, a temporally silent store,[7] or a speculative write update.

When the SCL protocol returns a value sooner than the coherence protocol, the computation using the value can overlap the coherence operation. Accurate SCL protocols hide coherence latencies, allowing the use of simpler but lower-performance coherence protocols without a commensurate performance penalty.

Coherence decoupling architecture

To support coherence decoupling, the system architecture must

- split, breaking a memory operation into a speculative read and a coherence operation;
- compute, providing mechanisms to execute with the speculative value; and
- recover, supporting an incorrect speculation detection and recovery.

The first task, splitting a memory operation into two suboperations, is straightforward. For the second task, the same mechanisms that support other forms of speculative execution can support speculative computation, although the growing coherence latencies require mechanisms that can buffer speculative state across hundreds to thousands of instructions. For the third task, the recovery mechanism buffers the speculative value in an MSHR, compares it with the value the coherence protocol returned, and recovers if a speculation is incorrect.

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 using the speculative value will not violate the memory consistency model. As Martin et al. observed, implementing value speculation correctly requires the same hardware as that used for aggressive implementations of sequential consistency.[8] This hardware support permits correct implementation of coherence decoupling without violating the memory consistency model.

SCL protocols for coherence decoupling

Backed by a simple and easily verifiable coherence protocol, many different performance-improving SCL protocols are possible. Each SCL protocol has a read component and an update component (which might 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 that are also in an invalid state. Table 1 summarizes the SCL protocols that we evaluated. Many other protocols are possible; evaluations for some appear in the literature.[9]

SCL protocol read component. The first read component policy simply returns the value in the local cache if the block is present (that is, if the invalid cached tag matches that of the load) even though the coherence state is invalid. We call this basic coherence decoupling (CD). This protocol speculates correctly if the accessed word is falsely shared or if it is 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, in which F stands for filter, adds a program counter-indexed confidence predictor to throttle low-confidence speculations. It reduces the number of times the system uses speculation

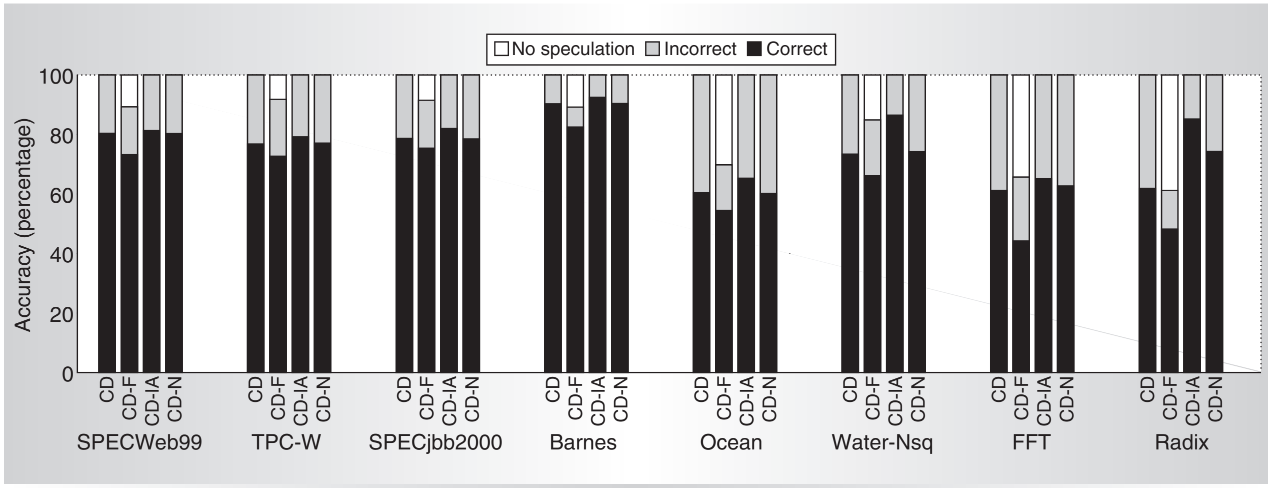

Figure 2. Coherence decoupling accuracy.

but improves the average speculation accuracy over that of the CD protocol.

In general, an SCL protocol's read component could return a value from an invalid (or valid) line anywhere in the system. The protocol's 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 chip multiprocessors. In this work, we consider flat, snooping symmetric multiprocessors (SMPs) only, because our simulation infrastructure cannot evaluate large-scale or hierarchical systems.

SCL protocol update component. An SCL protocol includes an update component to improve the speculation accuracy for truly shared data. SCL updates achieve this benefit by writing updates to invalid lines around the system. The extra bandwidth consumed is a trade-off for increased speculation accuracy.

A simple update policy, CD-IA (where IA stands for update at all things invalidated), 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 an invalid state, not just those transitioning from a shared to an invalid state. The other update policy, CD-N, broadcasts the dirty line after the writer has made *N* writes. Such updates require additional messages.

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

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.

Coherence decoupling accuracy

Figure 2 shows the ratio of correct to incorrect CD speculations (for all coherence misses) using a 4-Mbyte 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 five writes to a line.

All SCL protocols except for CD-F speculate on every coherence miss. CD-F uses the confidence filter to avoid issuing speculations that are likely incorrect. The base CD protocol makes more correct speculations than CD-F, but at the expense of more mispredictions. However, because of silent stores and false sharing, this simple protocol provides accuracy levels approaching those of many update protocols. For three commercial benchmarks

Table 2. Speedups for coherence decoupling protocols over traditional protocols (as a percentage), by policy.

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**

and Barnes, the base CD protocol can predict correct values for more than 70 percent of coherence misses. SCL Update protocols may occasionally lose accuracy by sending an update too early. For example, a speculative update may write a new value into an invalid line, following which the writer changes its local value back (a temporally silent store) but does not broadcast the update. In this case, the speculative reader will read the (now incorrect) first update and not the correct second update, whereas in a non-update protocol the reader would have read the original value, which would also have resulted in a correct speculation. Overall, coherence decoupling appears more accurate 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.

Coherence decoupling timing results

Table 2 shows the speedups over the baseline system (that is, the simple invalidation protocol with no coherence decoupling or speculation). We modeled a flushing mechanism to recover from misspeculations. The mechanism flushes all instructions younger in program order than the misspeculated one when the violation is detected (a *rolling flush*) rather than waiting until the violation reaches the head of the reorder buffer.

Table 2's rightmost column places an upper bound on the performance of coherence decoupling in the simulated system. This model treats all cache accesses that would have been coherence load misses as hits. SPECWeb99 and Ocean show large ideal benefits (34.6 and 34.5 percent), but Barnes shows a mere 1.4 percent because of its negligible level-two cache miss rates.

Coherence decoupling accuracy rates are high, partially or fully tolerating a third to one half of coherence misses. The speedups reflect those results for several benchmarks; SPECjbb2000 in particular reaches more than 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 percent, which is more than a quarter of the ideal speedup. In larger-scale systems (and particularly cache-coherent, nonuniform memory access 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.

Coherence decoupling is one of a set of new microarchitectural techniques that ease the burden on the parallel programmer, in addition to improving performance. Future work in this area focuses on two main directions: improving SCL protocols to further increase speculation accuracies and enabling more efficient recovery from misspeculations. Another open direction is the utility of coherence decoupling for both hierarchical multiprocessors and multiprocessors with directory-based cache coherence. MICRO

Acknowledgments

This work was supported in part by the US Defense Advanced Research Projects Agency under contracts NBCH30390004 and F33615-03-C-4106; the National Science Foundation under grants EIA-0071924, CCR-0311572, and CCR-9985109; the Intel Research Council; an IBM Faculty Partnership Award; and the University of Wisconsin Graduate School.

References

1. D. Lenoski et al., "The Stanford DASH Multiprocessor," *Computer*, vol. 25, no. 3, Mar. 1992, pp. 63-79.
2. A.R. Lebeck and D.A. Wood, "Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors," *Proc. 22nd Int'l Symp. Computer Architecture*

(ISCA 95), IEEE CS Press, 1995, pp. 48-59.

3. R. Rajwar and J.R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *Proc. 34th Int'l Symp. Microarchitecture* (Micro 34), IEEE CS Press, 2001, pp. 294-305.
4. K. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory," *Proc. 17th Int'l Symp. Computer Architecture* (ISCA 90), IEEE CS Press, 1990, pp. 15-26.
5. M.M.K. Martin, M.D. Hill, and D.A. Wood, "Token Coherence: Decoupling Performance and Correctness," *Proc. 30th Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, 2003, pp. 182-193.
6. M.H. Lipasti, C.B. Wilkerson, and J.P. Shen, "Value Locality and Load Value Prediction," *Proc. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 96), ACM Press, 1996, pp. 138-147.
7. K.M. Lepak and M.H. Lipasti, "Temporally Silent Stores," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 30-41.
8. M.M.K. Martin et al., "Correctly Implementing Value Prediction in Microprocessors that Support Multithreading or Multiprocessing," *Proc. 34th Int'l Symp. Microarchitecture* (Micro 34), IEEE CS Press, 2001, pp. 328-337.
9. J. Huh et al., "Coherence Decoupling: Making Use of Incoherence," *Proc. 11th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 04), ACM Press, 2004, pp. 97-106.

Jaehyuk Huh is a PhD student in computer science at The University of Texas at Austin. His research interests include microarchitecture, multiprocessors, and operating systems. Huh has a BS from Seoul National University, Korea, and an MS from the Department of Computer Sciences at The University of Texas at Austin. He received an IBM PhD research fellowship in 2004.

Doug Burger is an associate professor of computer sciences at The University of Texas at Austin. His research interests include high-performance, power-efficient, technology-scalable microprocessors. Burger has a BS from Yale University and an MS and a PhD from the University of Wisconsin, all in computer science. He is an Alfred P. Sloan Foundation Fellow, a senior member of the IEEE, and a member of the ACM.

Jichuan Chang is a PhD student in computer sciences at the University of Wisconsin-Madison. His research interests include multiprocessor cache coherence and speculation in chip multiprocessors. Chang has a BS and an MS from Beijing University, both in computer sciences.

Gurindar S. Sohi chairs the Computer Sciences Department at the University of Wisconsin-Madison. His research interests include the design of high-performance computer systems. Sohi has a BE in electrical and electronics engineering from the Birla Institute of Science and Technology, Pilani, India, and an MS and a PhD in electrical and computer engineering from the University of Illinois. 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.