Appears in the 11th International Conference on Architectural Support for Programming Languages and Operating Systems

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

This paper explores a new technique called *coherence decoupling*, which breaks a traditional cache coherence protocol into two protocols: a Speculative Cache Lookup (SCL) protocol and a safe, backing coherence protocol. The SCL protocol produces a speculative load value, typically from an invalid cache line, permitting the processor to compute with incoherent data. In parallel, the coherence protocol obtains the necessary coherence permissions and the correct value. Eventually, the speculative use of the incoherent data can be verified against the coherent data. Thus, coherence decoupling can greatly reduce — if not eliminate — the effects of false sharing. Furthermore, coherence decoupling can also reduce latencies incurred by true sharing. SCL protocols reduce those latencies by speculatively writing updates into invalid lines, thereby increasing the accuracy of speculation, without complicating the simple, underlying coherence protocol that guarantees correctness.

The performance benefits of coherence decoupling are evaluated using a full-system simulator and a mix of commercial and scientific benchmarks. Our results show that 40% to 90% of all coherence misses can be speculated correctly, and therefore their latencies partially or fully hidden. This capability results in performance improvements ranging from 3% to over 16%, in most cases where the latencies of coherence misses have an effect on performance.

Categories and Subject Descriptors: C.1.2 [Processor Architectures]: Multiple Data Stream Architectures (Multiprocessors), B.3.2 [Memory Structures]: Design Styles – Shared memory, C.4 [Performance of Systems] – Design studies

General Terms: Performance, Design, Experimentation, Measurement

Keywords: Coherence decoupling, speculative cache lookup, coherence misses, false sharing

ASPLOS'04, October 9–13, 2004, Boston, Massachusetts, USA.

1. INTRODUCTION

Multiprocessing and multithreading are becoming ubiquitous, even on single chips, despite growing system-wide communication latencies. Inter-processor communication in a shared-memory multiprocessor is carried out using a cache coherence protocol that enables the correct sharing of data among the multiple processors. Since the cache coherence protocol is a primary contributor to the latency of inter-processor communication, its design is arguably the most important aspect when designing a shared-memory multiprocessor. These protocols have been heavily studied by researchers for decades.

Reductions in average communication latencies can be achieved by tuning coherence protocols for specific communication patterns and/or applications. However, these optimizations for specific cases add to the complexity of the protocol, since the protocol must also ensure the correctness of data sharing in each specific case. Prior research has shown that large reductions in average communication latency are possible, but at the cost of protocols and systems that are too complex to be feasible. A competing approach requires the application programmers to tune their applications to work well with simpler protocols — for example, padding all data structures to reduce false sharing. This solution is equally problematic as it decreases parallel programmers' productivity considerably.

An important trend in computer architecture in the past two decades has been the use of speculation: rather than incurring the latency of waiting for the outcome of an event, the outcome is predicted, allowing execution to proceed with the prediction. The prediction is verified when the outcome of the event is known, and corrective action is taken if the prediction was wrong. Speculative execution has been successfully used to overcome performance hurdles in a variety of scenarios, for example, branch instructions (control speculation) [34, 41], ambiguous dependences (dependence speculation) [30], parallelization (speculative parallelization) [40], and locking overheads (speculative lock elision) [35].

In this paper we propose a technique called *coherence decoupling*, which applies speculation to the problem of long-latency shared-memory communication. This technique reduces the effect of these latencies, but neither exacerbates the programmer's task nor makes correctness of the coherence protocol more difficult to ascertain. Coherence decoupling breaks the communication of a shared value into two constituent parts: (i) the acquisition and use of the value, and (ii) the communication of the coherence permissions that indicate the correctness of the value and thus the exe-

cution. In traditional cache coherence protocols, these two aspects of communication have been merged into a single protocol; obtaining the coherence permissions must strictly occur before use of the data, thus serializing the two. Coherence decoupling enables separate protocols for the speculative use and eventual verification of the data. A *Speculative Cache Lookup* (or SCL) protocol provides a speculative value as quickly as possible, while in parallel the *coherence protocol* executes and eventually produces the correct value along with the requisite access permissions.

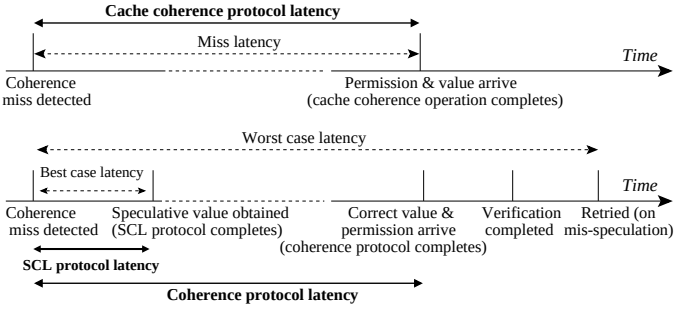

Figure 1: Coherence Decoupling

Separating the SCL protocol and the coherence protocol allows each to be tuned independently. This capability enables novel optimizations that permit higher performance with less complexity than traditional protocol optimizations. The separation also allows the two to be overlapped. The top part of Figure 1 shows the timing of events, with a conventional coherence protocol, for a read to a cache line that requires a change in coherence permissions. The cache coherence operation is followed by the arrival of the line with the correct data and the appropriate permissions, at which point the data can be used. The bottom part of Figure 1 shows the timing of events in a system with coherence decoupling. Here the SCL protocol could speculatively return the data earlier, for example, if a tag match occurs in a local cache (even if the line is invalid), while simultaneously launching the invalid-to-shared upgrade via the coherence protocol. When the coherence protocol returns the permissions and the correct value, the value is compared to the value returned by the SCL protocol. If the values 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 SCL and coherence protocol values differ, a full or partial 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 benefit of a successful speculation, and the cost of recovery.

In this paper, we explore different SCL protocols with varying speculation accuracies, while maintaining a simple, invalidation-based coherence protocol for correctness. The first SCL protocol we explore merely accesses the first value it finds (e.g., in the local cache) for which the tag matches, regardless of the coherence state of the line. This protocol greatly reduces performance losses due to false sharing, since falsely-shared values will not be changed by the owner of the line, thus guaranteeing successful speculation.

The next set of SCL protocols we explore are variants of a *write-update* protocol. This series of protocols trades off the accuracy of the values provided by the SCL protocol with the extra traffic used to distribute speculation-improving write values. Unlike a canonical cache-coherent, write-update protocol, which suffers from correctness problems and race conditions, variants of write-update SCL protocols are simple and correct. These SCL protocols change data only in invalid lines to maximize the chances of speculative cache lookups being correct, and since they are speculative, they can roll back on any error.

In Section 2, we describe related work in multiprocessor speculation, describing how coherence decoupling extends prior work in several new directions. In Section 3, we describe the coherence decoupling framework, correctness issues, and the set of SCL protocols that we study in this paper. In Section 4, we describe the simulation environment that we use, profile shared-memory benchmark miss patterns to identify the performance potential of coherence decoupling, and evaluate the performance of a number of SCL protocols. We conclude in Section 5 with a discussion of why we believe that coherence decoupling will be an important technique for future processors and systems.

2. PRIOR WORK

The prior research most relevant to coherence decoupling falls into three broad categories: (1) customized coherence protocols, (2) speculative coherence operations, and (3) speculation on the outcome of events in a multiprocessor execution.

Customized coherence protocols attempt to specialize underlying coherence protocols to reduce communication and coherence latencies for special cases. The Stanford Dash multiprocessor [23] included directory protocol optimizations for specific sharing patterns, as did the pairwise sharing protocol in the Scalable Coherent Interface [15], and migratory sharing protocols [3, 4, 42]. Other proposed protocols adapt to different sharing patterns [5, 43], or trade-off write-invalidate and write-update protocols [2, 31, 37].

A different approach exposes coherence protocols to software; cooperative shared memory [14] used software directives to allow applications to guide the coherence protocols with check-in and check-out primitives. Exposure to software reached its zenith with the Stanford Flash [19] and Wisconsin Tempest/Typhoon [38], which enabled software to customize coherence protocols on a per-application basis [7]. These research directions were discontinued as it became apparent that protocol customization was too difficult for most programmers.

Speculative coherence operations are in a sense the converse of the coherence decoupling approach. Coherence decoupling performs speculative *computation* to reduce the need for extra coherence operations or optimizations. Speculative coherence operations instead speculatively initiate extra coherence messages (e.g., invalidates or upgrades) in the base protocol to reduce the latency of shared accesses and thus the need for speculative computation. Lebeck and Wood proposed Dynamic Self Invalidation [22], in which processors speculatively flush their blocks based on access history, reducing the latency of later invalidations by remote writers. Mukherjee and Hill proposed a "coherence message predictor" [32] that initiated coherence messages speculatively. Kaxiras and Goodman extended this approach with PC-indexed tables rather than address-indexed tables [17].

Lai and Falsafi restricted these tables to holding patterns of memory demand requests only [20], thus providing a more effective predicted stream of coherent block read requests. They also replaced the access counts used in Dynamic Self-Invalidation with two-level adaptive prediction in a "last-touch predictor" [21]. Finally, Kaxiras and Young explored a range of policies to predict the set of sharers of a given cache line [18], as did Martin et al. [26] with "destination set prediction." Martin et al. also proposed token coherence [27] which, like coherence decoupling, breaks a protocol into separate performance and correctness protocols, but which does not employ speculation to overlap computation and coherence operations.

The prior work most similar to coherence decoupling fits into two categories. The first category is *speculative synchronization*, in which the outcome of a synchronization event is speculated. For example, a lock is speculated to be unheld, permitting entry into critical sections [29, 35, 36]. The similarity to coherence decoupling is that both techniques employ speculative access to shared variables. With speculative synchronization, however, speculation is limited to locks only. Temporally silent stores and the MESTI protocol [25], a proposed alternative to speculative synchronization, exploits the predictable behavior of the values of lock variables to reduce the coherence protocol overhead in a lock handoff, but is neither a speculative protocol, nor does it launch speculative operations.

The second category of event outcome speculation techniques use speculation to overcome the performance limitations of strong memory models [9, 12, 33, 44]. These techniques speculate that a memory model (e.g., sequential consistency) will not be violated if memory references are executed in an optimistic fashion. Memory operations that have been carried out optimistically are buffered and these buffers are checked to see if the optimistic execution has resulted in a possible violation of the memory consistency model [10]. Execution is rolled back in case of a violation. This form of speculative execution is widely used in commercial multiprocessors today, but is significantly less aggressive than coherence decoupling.

3. COHERENCE DECOUPLING

Coherence decoupling separates a cache coherence protocol into two parts: (i) a speculative cache lookup (SCL) protocol, which returns a speculative value that can be used for further computation, and (ii) a coherence protocol, which returns the correct value (as defined by the memory consistency model) and the requisite permissions to use the value. If the SCL protocol can return a value faster than the coherence protocol, the computation using the value and the coherence operations can be overlapped. Higher accuracy in the SCL protocol allows for more frequent hiding of coherence protocol latencies, allowing simpler but lower performance coherence protocols to be used without a commensurate performance penalty.

We consider how to support coherence decoupling (Section 3.1), how to ensure correctness in a system with coherence decoupling (Section 3.2), and present some SCL protocols for coherence decoupling (Section 3.3).

3.1 Coherence Decoupling Architecture

To support coherence decoupling the system architecture must: (i) *split*, providing a means to split a memory operation into a speculative load operation and a coherence operation, (ii) *compute*, providing mechanisms to support execution with speculative values, and (iii) *recover*, providing a means for detecting a mis-speculation and recovering correctly from it.

Splitting a memory operation (*i* above) into two sub-operations is straightforward, as is the recovery process (*iii* above) of comparing the results of the speculative load operation and the coherence operation to detect a mis-speculation. The speculated value may be buffered in an MSHR, which then compares the value against the correct value when the coherence protocol returns the cache line.

To support speculative computation (*ii* above), the same mechanisms that are used to support other forms of speculative execution can be used. Since coherence latencies are growing to hundreds of cycles, however, current microarchitectural mechanisms to support in-processor speculation (e.g., branch speculation) are likely to be inadequate. Mechanisms that can buffer speculative state across hundreds to thousands of speculative instructions will be necessary. Examples include the Address Resolution Buffer (ARB) [8] or the Speculative Versioning Cache (SVC) [13] used for Multiscalar processors, load and store buffers used for speculatively improving the performance of sequential consistency (SC) [12, 33], and statically allocated, kilowindow reservation stations in the TRIPS architecture [39].

In this paper, for recovering from mis-speculations, we model the standard recovery policy for techniques that use deep speculation: squashing the offending instruction and all succeeding instructions.

3.2 Correctness of Coherence Decoupling

As Martin et al. have observed [28], implementing value speculation correctly requires hardware that performs the same function as that used for aggressive implementations of sequential consistency (SC) and vice versa.

Coherence decoupling relies upon the above observation for correctness. Obtaining a value speculatively with 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, and using the coherence protocol to verify the speculation. Thus, if we use the same hardware to implement coherence decoupling that we use to implement aggressive implementations of SC, coherence decoupling can be implemented without any correctness implications for the memory consistency model.

3.3 SCL Protocols for Coherence Decoupling

A wide range of SCL protocols for coherence decoupling are possible. Although the SCL protocols can be combined with arbitrarily-complex coherence protocols, coherence decoupling enables these aggressive SCL protocols to be backed by a simple, easily-verifiable coherence protocol. In this work we therefore measure only a simple invalidation-based coherence protocol, described in Section 4 and rely on the SCL protocols to improve performance.

An SCL protocol has two components. The first is the *read component* — the policy for obtaining the speculative value (i.e., where the protocol searches for a speculative value to use). The second is the *update component*, in which the SCL protocol may speculatively send writes to invalid cache lines (former sharers) to increase the probability that a subse-

SCL Component	Policy	Description
Read	CD	Use the locally cached incoherent value for every L2 miss
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-C	Use invalidation piggyback if the value is special (compressed)
Update	CD-N	Update all sharers after N writes to a block (N=5 in Section 4)
Update	CD-W	(Ideal): Update on every write if any sharers exist

Table 1: Coherence Decoupling Protocol Components

quent coherence decoupled access will read the correct value. This component trades increased bandwidth — consumed by sending speculative writes around the system — for improved speculation accuracy. The update component may be null in some SCL protocols.

3.3.1 SCL Protocol Read Component

The first policy for the read component we propose simply uses the value in the local cache if the block is present (i.e., the tag matches) and if the block is either in an invalid state or in a shared state for an atomic load access. We call this CD, for basic coherence decoupling.

Since CD speculates on the value of every load operation that finds a matching tag (but with the wrong permission), it may incur a large number of mis-speculations, triggering too many rollbacks. The next SCL read component policy we propose, called "Coherence Decoupling + Filter" (or CD-F), employs a confidence mechanism — a PC-indexed table of counters — to throttle speculations. For some extra hardware, CD-F reduces the number of times speculation is employed (i.e., it reduces the *coverage*), thereby decreasing the total number of mis-speculations, but improving the average speculation *accuracy* over the base CD protocol.

In general the read component of an SCL protocol could return a (possibly incorrect) value from anywhere it finds in the system, if the latency of doing so is sufficiently lower than the latency of accessing it through the coherence protocol. In a directory-based cache coherent machine, for example, the SCL protocol could first access the local cache and then the home memory of the invalid line, using the invalid data while the home directory communicated with an exclusive owner of the block. In another example, the value could reside in a geographically-proximate cache in a hierarchical multiprocessor (e.g., another cache on the same chip in a multiprocessor built from CMPs). In this paper, however, we consider only a flat symmetric multiprocessor leaving the issue of SCL protocols for hierarchical systems to future work.

An SCL protocol with only a read component (and a null update component) speculates correctly if the contents of the accessed word in the invalid block have not changed remotely since being invalidated (false sharing [6]), have been changed remotely to the same value (silent stores [24]), or have been changed remotely to a different value and then changed back to the original value (temporally silent stores [25]). This capability allows the problem of false sharing to be greatly mitigated. As long as there is sufficient work for the processor to do after it speculates on falsely-shared data, the coherence protocol latency for such a request can be overlapped completely. A successful CD protocol will prevent the programmer from having to recode data structures to reduce false sharing (if they can even figure out that false sharing is occurring in the first place).

3.3.2 SCL Protocol Update Component

We can further attempt to improve the accuracy of speculation for truly-shared data by adding *update components* to the SCL protocol. An update component speculatively sends updated data around the system and writes them into invalid cache lines. The update component of an SCL protocol thus trades increased speculation accuracy for the extra bandwidth consumed by the updates.

A variety of protocols for the update component of an SCL protocol, with different accuracies and bandwidth requirements, are possible. We present several such protocols in this section; it is easiest to view them as variants of a basic write-update protocol. It is important to note that since these updates are speculative, they can be completely non-blocking for the writer and can proceed in parallel with other operations. If a speculative write finds a copy of the line which is not in invalid state, the write is simply dropped and correctness preserved. This capability is in contrast to a canonical write-update cache coherence protocol which requires the writer to view the transmission of the write updates as a blocking operation.

Our first update component for an SCL protocol, CD-IA, piggybacks the value created by the writer along with the invalidation message used to invalidate remote caches. The message size is increased to include a data packet in addition to the address packet. However, since we model a bus-based broadcast coherence protocol in this paper, CD-IA updates the data in all caches which have the block (i.e., caches already in an invalid state) and not only the sharing caches that need to be invalidated.

CD-C is a variant of CD-IA; it uses compressed updates to reduce the message overhead. For the commercial workloads studied in this paper, many of writes that result in an invalidation message frequently write the values 0, 1, or -1. CD-C piggybacks updates for only these values to the initial invalidation message, allowing these updates to be communicated to remote caches by adding only two additional bits to the invalidation message.

The remaining protocols for the SCL update component that we consider also send updates after the initial invalidations have been sent. Consequently these additional updates require additional messages. CD-N broadcasts the dirty line after N updates have been made by the same writer. Other possible policies might broadcast the block after *every* N writes, or broadcast the block after the (predicted) last write to the block. With the bus-based interconnect that we model, which has limited bandwidth, these policies performed much worse than the others, so we do not present their results in this paper. They may be more compelling on higher-bandwidth topologies, which we leave to future work.

Finally, CD-W is an ideal policy that sends an update on every write, if invalid sharers exist. That is, it uses a conventional write-broadcast protocol for the update component of

Feature	Parameters
Issue width	4
Window size	512-entry RUU
Number of CPUs	16
L1 cache	split I/D, 128K, 4-way, 128-byte block
L2 cache	unified, 4M, 4-way, 128-byte block
MSHR size	32
Base protocol	bus-based MOESI
Bus bandwidth	12.8GB/s
L1/L2 hit latencies	2 cycles / 24 cycles
Memory access latency	460 cycles
Cache-to-cache latency	400 cycles

Table 2: Simulated machine configuration

the SCL protocol. In a machine with directory-based cache coherence, the writer could maintain the list of sharers after invalidations for propagating occasional writes, or the system could use destination set prediction for guessing which nodes hold invalid copies of a line [18, 26].

Table 1 summarizes the SCL protocol components that we consider in this paper. Note that the read component of the SCL protocol is orthogonal to the update component of the SCL protocol. Thus either of the read components (`CD` or `CD-F`) could be used in conjunction with any of the update components (`CD-IA`, `CD-C`, `CD-N`, or `CD-W`), or even with a null update component. To reduce the number of combinations, however, for the remainder of the paper when we discuss an SCL protocol with a non-null update component (`CD-IA`, `CD-C`, `CD-N`, or `CD-W`), we will assume that it uses `CD` for its read component.

Clearly other options for speculatively passing around data are possible, trading off speculation accuracy with message bandwidth. Existing cache coherence protocol optimizations for *correctly* passing data can be leveraged into have *speculative* versions. For example, we could have speculative competitive write-update protocols, or speculative customized protocols that can dynamically learn the communication pattern of an application and try to optimize the data communication. Such protocols are left for future work.

4. RESULTS

We ran our experiments on *MP-Sauce*, an execution-driven, full-system multiprocessor timing simulator derived from IBM's SimOS-PPC, which uses AIX 4.3.1 as the simulated OS. Full-system simulation is necessary since many of our workloads (commercial benchmarks) interact with the OS substantially. In the simulator itself, the architectural timing code extends the SimpleScalar processor timing model with multiprocessor support, including a recoding of the processor core to be fully execution driven. Network contention due to speculative updates is also modeled (except for the ideal `CD-W` protocol). Table 2 lists the most relevant machine parameters from the simulated system.

We simulated three commercial applications and five scientific shared-memory benchmarks from the SPLASH2 suites. The three commercial workloads are TPC-W using a MySQL backend running on Apache, SPECWeb99 running on Apache, and SPECJbb using the IBM Java virtual machine. The SPLASH applications we simulate are Barnes, Ocean, Watersq, FFT, and Radix.

Since multi-threaded, full-system simulations produce results that vary from run to run, we replicated the methodology in other studies and ran each experiment multiple times, injecting small timing variations [1]. We report the mean of the execution time across the multiple runs as our experimental result.

In this paper, we limit our simulations to 16-node SMP systems, for two reasons. First, small-scale hierarchical (CMP-based) NUMA systems are still emerging, although they certainly provide opportunities for coherence decoupling. Second, more traditional, directory-based CC-NUMA multiprocessors are typically too large to simulate in our full-system environment—the operating system that we simulate can support configurations only up to 24 processors. We expect that the relative performance benefits we show will only increase in larger-scale systems, where coherence misses are more frequent and latencies are longer.

4.1 Microbenchmarks

To understand the effectiveness of coherence decoupling, we show the results of two simple microbenchmarks, which are designed to generate false sharing misses. `simple-fs` loads falsely shared data, while executing both dependent and independent instructions every loop iteration. The ratio of dependent and independent instructions is set to 1:3. The dependent instructions are simple additions and multiplications, which use the value returned from a load that incurred a false sharing miss. `critical-fs` generates a false sharing miss on each iteration, but calculates the address of that load using the value returned from the false sharing miss of the previous loop iteration. Figure 2 presents the key microbenchmark code fragments on the left half of the figure.

The right half of Figure 2 shows the microbenchmarks' normalized execution times, varying cache-to-cache communication latencies from 200 to 1000 cycles, with each microbenchmark using both the baseline and the `CD` protocol. `simple-fs` has speedups from 12% to 17% over the base case, as communication latencies increase. With a 512-entry RUU, `CD` can execute approximately 120 dependent instructions, none of which can be executed in the baseline system until the falsely shared data return. Despite this additional latency tolerance, however, `CD` can not hide communication latencies past a certain size due to the finite instruction window size (512 entries), after which point performance degrades more quickly as latencies increase.

`critical-fs` forces a data dependence between two loads, placing consecutive false sharing misses on the critical path of execution. Since the delay to calculate addresses and issue the subsequent loads can not be tolerated in this microbenchmark, false sharing misses have a major effect on performance. `CD` can calculate the next address by using

```
for(i = 0; i < MAX; i++) {
   fval = array[i].value;  /* false sharing miss */
   dep_val1 = fval + 2;
   indep_val1 = local + 2;
   dep_val2 = fval * 3;
          …
   /* dependent and independent instructions */
}
```

(a) simple-fs

```
for(i = 0; i < MAX; i++) {
   index = array[index].value; /* false sharing miss */
   sum += index;
}
```

(b) critical-fs

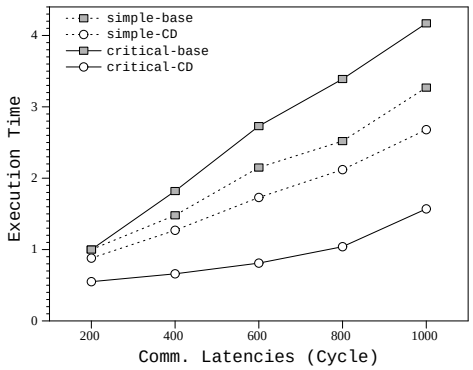

Figure 2: Microbenchmarks

data in local invalid blocks to issue the subsequent memory accesses early, overlapping false sharing misses. Even at 200 cycles, the speedup of CD is more than 45%. As communication latencies increase, the performance of CD degrades much more slowly than the baseline. At a 1000-cycle communication latency, the baseline system is about three times slower than CD, although the slopes of performance degradation become similar once the finite window and MSHR sizes prevent tolerance of longer latencies.

4.2 Miss Profiling Results

Figure 3 shows a breakdown of L2 read misses for a number of cache configurations (1MB and 4MB capacity, and 128-byte and 512-byte cache lines). We partition read misses into coherence misses and "other" misses, which include capacity, conflict, and compulsory misses. We define coherence misses as having a matching tag in the L2 cache but the wrong coherence state (e.g., invalid state on a read), thus requiring remote communication. We subdivide coherence misses into false sharing, silent stores, and true sharing misses. Coherence misses are counted as true sharing misses only when the correct values differ from those in the local cache's stale copy. Silent stores update a cache block, but the values have not been changed from the old values stored in the local invalid block. The silent store bar in the figure includes both temporally silent stores and silent stores. For false sharing misses and silent stores, local cache lines in invalid state will have the correct values. The sum of silent stores and true sharing misses are the traditional address-based true sharing misses by Dubois' definition [6].

The data show that the fraction of coherence misses is significant for every one of the commercial benchmarks (a minimum of 12% in FFT). The data corroborate the expected trend that as the cache size grows — for a fixed size workload — the coherence misses increase, to 80% in SPECWeb, 81% in TPC-W, and 67% in SPECJbb. Given the enormous cache sizes in future server-class systems (36MB per processor die in IBM's Power5 system), we expect that coherence misses will be a significant and growing component of communication in future multiprocessor systems.

A significant fraction of L2 load misses across the benchmarks are coherence misses caused by silent stores, from 7% (Barnes) to 16% (Ocean). The base CD protocol will predict the correct value for the silent stores as well as for false sharing misses. The ratio of false sharing misses (plus silent stores) to true sharing misses increases as the block sizes increase, for all benchmarks. Of those being simulated, the cache configuration with the lowest average miss rate (4MB with 512B blocks) shows that from 13% (FFT) to 71% (Barnes) of all L2 misses result from false sharing. If cache size growth outstrips working set size growth, as is certain for some benchmarks, coherence misses in general and false sharing in particular will increase as a fraction of L2 misses.

4.3 Coherence Decoupling Accuracy

In this section, we present speculation accuracies for a number of the coherence decoupling policies that are described in Section 3. Figure 4 shows the ratio of correct to incorrect CD speculations (for all coherence misses) using a 4MB cache with 128-byte blocks, for a subset of the policies described in Section 3. 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 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 misspeculation.

Update-component protocols have better accuracy than CD for some benchmarks, but the improvement is modest. CD-W improves the prediction accuracy for FFT and Radix, but it does not increase the accuracy for the other benchmarks. The CD-IA policy sees better accuracy for Water-nsq, FFT, and Radix, than CD-C, because the latter policy can not deliver a truly-shared value, if the value is not one of the values that can be encoded and sent along with the invalidation (-1, 0, or 1). 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.

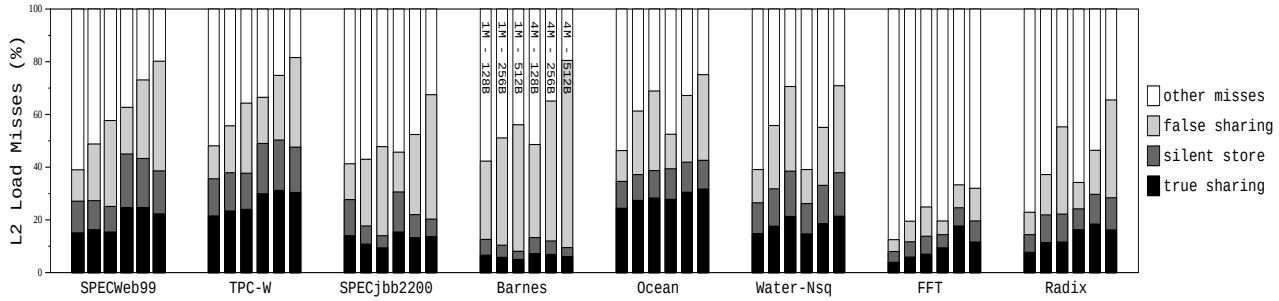

Figure 3: L2 Load Miss Breakdowns.

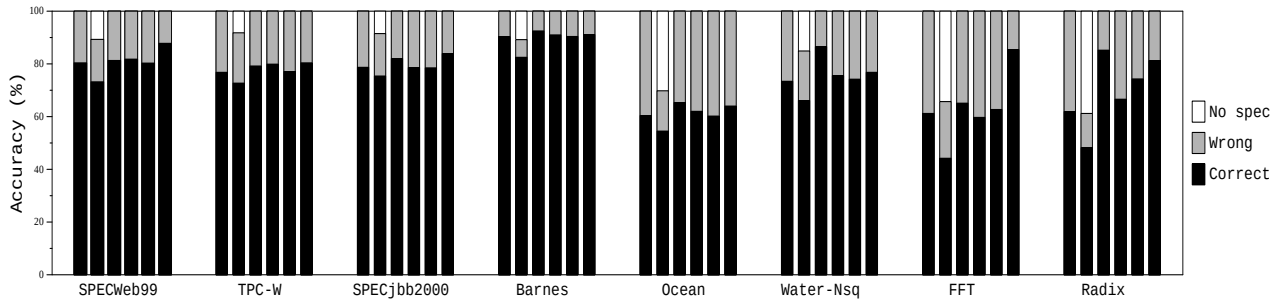

Figure 4: Accuracy of Coherence Decoupling (from left to right: CD, CD-F, CD-IA, CD-C, CD-N, CD-W)

4.4 Coherence Decoupling Timing Results

We now consider timing simulation results for a system with coherence decoupling. Table 3 shows the speedups over the baseline system (which is the simple invalidation protocol with no coherence decoupling or speculation) for the range of policies described in Section 3. 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 rolling flush mechanism reduces the cost of speculation recovery, and is implemented in modern server processors such as IBM's Power5 [11].

The right-most column of Table 3 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. For TPC-W, the best-case speedup is 17.8%, providing only moderate opportunities for coherence decoupling speedups. SPECWeb99 and Ocean show larger benefits (34.6% and 34.5%). The only benchmark with an ideal coherence decoupling speedup of under 15% is Barnes, which is 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.

The update-based SCL protocols consume extra network bandwidth to increase prediction accuracies. Table 4 presents the network bandwidth overhead for `CD-IA` and `CD-N5`. In all update protocol experiments, we only transmit the updated words (not entire cache lines) to reduce bandwidth consumed. `CD-IA` incurs only small traffic increases (under 4%). `CD-N5` incurs much larger increases in traffic, with a range of 6% to 30% except for Barnes, which increases traffic by 95%. Due to Barnes' low L2 miss rates, however, that outlier has little effect on performance.

4.5 Latency Tolerance Profiles

In Table 5, we show the breakdown of instructions issued for each benchmark during the "decoupling window", the time between when a load speculatively accesses invalid data in the cache and when coherence permissions return with the correct value, using the CD policy. Since separate CD loads may overlap, we count instructions issued for the first overlapped CD load; we did not measure instructions issued for the second (and other) overlapped CD loads.

The first row of the table shows the number of correct coherence decoupled operations per 1K instructions. The remaining rows of the table separate the instructions in the decoupling window into three categories: data-dependent instructions, control-dependent instructions, and independent instructions. Data-dependent instructions are simply instructions that use the result of the decoupled load directly or indirectly, through register or memory dependences. Control-dependent instructions (in this definition) are instructions that are issued correctly during the decoupling window because coherence decoupling permitted quicker resolution of a mis-predicted branch (data dependent on the CD load), thus allowing instructions down the correct path to issue more quickly than if the mis-predicted branch had waited for the CD load to complete. Independent instructions are simply instructions that are neither data dependent on the

Benchmark	CD	CD-F	CD-IA	CD-C	CD-N5	CD-W	Optimal
SPECWeb99	13.8%	11.0%	13.2%	13.1%	14.9%	18.0%	34.6%
TPC-W	1.2%	2.6%	2.3%	1.7%	1.4%	2.4%	17.8%
SPECjbb2000	16.6%	15.8%	13.5%	13.0%	17.1%	16.5%	26.3%
Barnes	0.6%	0.4%	0.7%	0.7%	0.8%	0.6%	1.4%
Ocean	6.9%	4.7%	8.2%	7.4%	6.0%	7.5%	34.5%
Water-Nsq	2.1%	1.7%	2.8%	3.5%	0.7%	5.4%	17.4%
FFT	5.1%	4.2%	6.1%	7.2%	4.6%	10.8%	21.4%
Radix	6.8%	3.6%	7.6%	8.8%	6.3%	12.0%	42.4%
Mean	6.6%	5.5%	6.8%	6.9%	6.5%	9.1%	24.5%

Table 3: Speedups for Coherence Decoupling

Benchmarks	CD-IA	CD-N5
SPECWeb99	3.6%	7.9%
TPC-W	3.9%	18.5%
SPECjbb2000	2.5%	2.5%
Barnes	2.7%	95.3%
Ocean	3.4%	6.1%
Water-Nsq	2.0%	28.1%
FFT	2.8%	10.5%
Radix	3.2%	8.2%

Table 4: Data Traffic Increase

load nor are issued past CD-accelerated recovery of data-dependent mis-predicted branches.

There are fewer data-dependent instructions (from 6 to 19) than independent instructions (73 to 232). Control-dependent instructions are more numerous for some benchmarks; SPECjbb2000 has 16 control-dependent instructions and Water-nsq has 21 instructions. CD loads in those benchmarks help to resolve correct execution paths early. The early resolution of mispredicted branches will grow in importance in future processors that employ deeper speculation. For L2 cache misses on single-threaded applications, a similar observation was made by Karkhanis et al. [16].

5. CONCLUSIONS

This paper considered the use of speculation to tolerate the long latencies of inter-processor communication in shared memory multiprocessors. The proposed approach, called *coherence decoupling*, breaks up the cache coherence protocol, which is used to implement coherent inter-processor communication, into a speculative cache lookup (SCL) protocol that returns a speculative value, and a coherence correctness protocol that confirms the correctness of the speculation. An early return of a (speculative) value allows further useful computation to proceed in parallel with the coherence correctness protocol, thereby overlapping long coherence latencies with useful computation. Furthermore, decoupling the SCL protocol, which returns a value, from the protocol that 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.

We implemented a variety of options for the two components of an SCL protocol: the read component and the update component. The basic read component returns the value from a matching invalid cache line for which the access permissions are not correct. Another option we measured was the addition of a confidence filter to determine when coherence decoupling should be employed, to reduce the number of mis-speculations. For the update component, we considered several variations of a canonical write-update protocol. These variations trade off the accuracy of speculation of the SCL protocol with the additional bandwidth required.

Using a full-system simulator built on PowerPC/AIX, and running a set of commercial workloads and scientific workloads, our experiments showed that coherence misses are a significant fraction of total L2 misses, ranging from 10% to 80%, and averaging around 40% for large caches. Coherence decoupling has the potential to hide the miss latency for about 40% to 90% of all coherence misses, mis-speculating roughly 20% of the time.

We also measured the performance benefits of coherence decoupling. Several of the benchmarks are sensitive to coherence misses, so lower coherence latencies can improve performance. On these workloads, coherence decouping was able to achieve modest improvements. One of the benchmarks is affected little by coherence misses and, unsurprisingly, coherence decoupling did not help in this case. These results suggest that coherence decoupling is generally able to overcome the performance drawbacks of false sharing and, furthermore, allow lower effective latencies even when true sharing is present.

We expect techniques like coherence decoupling to grow in importance for future processors and systems for several reasons:

- First, multiprocessors and/or multithreaded processors will be soon be ubiquitous; almost every future processing chip will employ some form of multiprocessing or multithreading. Rather than burdening programmers with having to reason about the performance effects of data sharing, architects can develop alternative techniques to overcome these performance impediments without burdening the programmer. Coherence decoupling is a technique that overcomes one such performance impediment (false sharing), and mitigates true sharing in some cases.

- Second, with increasing cache sizes, coherence misses will account for a larger fraction of all cache misses.

	SPECWeb99	TPC-W	SPECjbb2k	Barnes	Ocean	Water-nsq	FFT	Radix
Correct CD/1K inst.	6.64	2.00	1.48	0.38	1.37	0.64	0.70	1.65
Data dependent insts	6.5	7.4	7.1	9.0	6.1	18.6	5.2	6.7
Control dependent insts	10.2	12.8	15.8	13.0	4.7	20.7	7.2	5.7
Independent insts	73.7	93.7	126.1	100.1	189.9	220.4	215.2	232.8

Table 5: Executed Instructions During Coherence Decoupling (using the CD policy)

This trend, coupled with increasing communication latencies, will cause the performance loss due to coherence misses to become a larger fraction of the overall performance loss. The performance loss for coherence misses will be magnified even further as other sources of performance losses (e.g., locks) are attenuated (for example, with speculative synchronization).

- Third, as communication latencies grow, there will be temptation to make coherence protocols more complex to reduce average latency. We believe that coherence protocols should be kept simple, relying on microarchitectural techniques to reduce communication-induced performance losses. Again, coherence decoupling is such a technique: the SCL protocol can allow the latency of the coherence protocol to be overlapped with computation that is likely to be useful.
- Finally, much of the hardware support required to support coherence decoupling is very likely to exist for other reasons — e.g., to overcome the performance limitations of sequential consistency, or to implement other speculative execution techniques. This fact will permit coherence decoupling to be implemented with less additional hardware and complexity.

For future work we plan to study better SCL protocols to increase speculation accuracies further, as well as techniques for efficient mis-speculation recovery. We also plan to study the utility of coherence decoupling for 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. REFERENCES

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th Int. Symp. on High-Performance Computer Architecture*, pages 7–18, Feb. 2003.

[2] C. Anderson and A. Karlin. Two adaptive hybrid cache coherency protocols. In *Proceedings of the 2nd Int. Symp. on High-Performance Computer Architecture*, pages 303–313, Feb. 1996.

[3] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Int. Symp. on Computer Architecture*, pages 98–108, May 1993.

[4] F. Dahlgren. Boosting the performance of hybrid snooping cache protocols. In *Proceedings of the 22nd Int. Symp. on Computer Architecture*, pages 60–69, June 1995.

[5] F. Dahlgren, M. Dubois, and P. Stenström. Combined performance gains of simple cache protocol extensions. In *Proceedings of the 21st Int. Symp. on Computer Architecture*, pages 187–197, Apr. 1994.

[6] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th Int. Symp. on Computer Architecture*, pages 88–97, May 1993.

[7] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing*, pages 380–389, Nov. 1994.

[8] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory. In *Proceedings of the 17th Int. Symp. on Computer Architecture*, pages 15–26, May 1990.

[10] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Proceedings of the Third ACM Symp. on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[11] P. N. Glaskowsky. IBM Raises Curtain on Power5. *Microprocessor Report*, Oct. 14 2003.

[12] K. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Int. Symp. on Computer Architecture*, pages 162–171, May 1999.

[13] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings of the The 4th Int. Symp. on High-Performance Computer Architecture*, pages 195–205, Feb. 1998.

[14] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, 1993.

[15] IEEE. IEEE Standard for Scalable Coherent Interface (SCI), 1992. IEEE 1596-1992.

[16] T. Karkhanis and J. Smith. A day in the life of a cache miss. In *Proceedings of 2nd Annual Workshop On Memory Performance Issues*, May 2002.

[17] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *Proceedings of the 5th Int. Symp. on High*

Performance Computer Architecture, pages 161 – 170, Jan. 1999.

[18] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *Proceedings of the 6th Int. Symp. on High Performance Computer Architecture*, pages 156–167, Feb. 2000.

[19] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Int. Symp. on Computer Architecture*, pages 302–313, Apr. 1994.

[20] A.-C. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Int. Symp. on Computer Architecture*, pages 172 – 183, May 1999.

[21] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Int. Symp. on Computer Architecture*, pages 139–148, June 2000.

[22] 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*, pages 48–59, June 1995.

[23] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[24] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *Proceedings of the 33rd Int. Symp. on Microarchitecture*, pages 22–31, Dec. 2000.

[25] 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*, pages 30–41, Oct. 2002.

[26] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared memory multiprocessors. In *Proceedings of the 30th Int. Symp. on Computer Architecture*, pages 206–217, June 2003.

[27] 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*, pages 182–193, June 2003.

[28] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proceedings of the 34th Int. Symp. on Microarchitecture*, pages 328–337, Dec. 2001.

[29] J. F. Martínez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, Oct. 2002.

[30] A. I. Moshovos, S. E. Breach, T. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Int. Symp. on Computer Architecture*, pages 181–193, June 1997.

[31] F. Mounes-Toussi and D. J. Lilja. The potential of compile-time analysis to adapt the cache coherence enforcement strategy to the data sharing characteristics. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):470–481, May 1995.

[32] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th Int. Symp. on Computer Architecture*, pages 179–190, June 1998.

[33] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ILP processors. In *Proceedings of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, Oct. 1996.

[34] Y. N. Patt, W. M. Hwu, and M. Shebanow. HPS, a New Microarchitecture: Rationale and Introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*, pages 103–108, 1985.

[35] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Int. Symp. on Microarchitecture*, pages 294–305, Dec. 2001.

[36] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Oct. 2002.

[37] A. Raynaud, Z. Zhang, and J. Torrellas. Distance-adaptive update protocols for scalable shared-memory multiprocessors. In *Proceedings of the 2nd Int. Symp. on High-Performance Computer Architecture*, pages 323–334, Feb. 1996.

[38] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level Shared Memory. In *Proceedings of the 21st Int. Symp. on Computer Architecture*, pages 325–336, Apr. 1994.

[39] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the 30th Int. Symp. on Computer Architecture*, pages 422–433, June 2003.

[40] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th Int. Symp. on Computer Architecture*, pages 414–425, June 1995.

[41] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transaction of Computer*, 39(3):349–359, 1990.

[42] P. Stenström, M. Brorsson, and L. Sandberg. Adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Int. Symp. on Computer Architecture*, pages 109–118, May 1993.

[43] Q. Yang, G. Thangadurai, and L. Bhuyan. Design of adaptive cache coherence protocol for large scale multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 281–293, May 1992.

[44] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.