INVISIFENCE: Performance-Transparent Memory Ordering in Conventional Multiprocessors

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

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

Thomas F. Wenisch
University of Michigan
twenisch@eecs.umich.edu

Abstract

A multiprocessor's memory consistency model imposes ordering constraints among loads, stores, atomic operations, and memory fences. Even for consistency models that relax ordering among loads and stores, ordering constraints still induce significant performance penalties due to atomic operations and memory ordering fences. Several prior proposals reduce the performance penalty of strongly ordered models using post-retirement speculation, but these designs either (1) maintain speculative state at a per-store granularity, causing storage requirements to grow proportionally to speculation depth, or (2) employ distributed global commit arbitration using unconventional chunk-based invalidation mechanisms.

In this paper we propose INVISIFENCE, an approach for implementing memory ordering based on post-retirement speculation that avoids these concerns. INVISIFENCE leverages minimalistic mechanisms for post-retirement speculation proposed in other contexts to (1) track speculative state efficiently at block-granularity with dedicated storage requirements independent of speculation depth, (2) provide fast commit by avoiding explicit commit arbitration, and (3) operate under a conventional invalidation-based cache coherence protocol. INVISIFENCE supports both modes of operation found in prior work: speculating only when necessary to minimize the risk of rollback-inducing violations or speculating continuously to decouple consistency enforcement from the processor core. Overall, INVISIFENCE requires approximately one kilobyte of additional state to transform a conventional multiprocessor into one that provides performance-transparent memory ordering, fences, and atomic operations.

Categories and Subject Descriptors

C.1.4 Computer Systems Organization [*Processor Architectures*]: Parallel Architectures

General Terms

Design, Languages, Performance

Keywords

Memory Consistency, Parallel Programming

1. Introduction

Stalls due to memory ordering constraints in shared-memory multiprocessors can result in significant performance penalties [1, 5, 8, 12, 14, 15, 26, 28, 32, 33]. Such stalls arise not just because of ordering requirements among loads and stores but also from atomic operations and explicit memory ordering fences, which occur frequently in highly-tuned multithreaded applications due to these applications' usage of fine-grained locking and lock-free synchronization. Thus, even relaxed consistency models can incur significant performance penalties due to memory ordering [8, 31, 32, 33]. Stronger consistency models incur even larger delays.

To reduce this performance penalty, current processors employ in-window speculative memory reordering [13] and post-retirement store buffers [3] (FIFO or coalescing, depending on the consistency model). However, performance penalties remain because of limited capacity of FIFO store buffers (implemented as CAMs to support load forwarding) and/or latency of atomic operations and fences (typically implemented by stalling dispatch or commit until the store buffer drains). As Figure 1 shows, memory ordering constraints block instruction commit for a significant fraction of time not only for sequential consistency but also for consistency models that relax only store-to-load ordering (*e.g.*, SPARC's TSO) and even for models with fully relaxed ordering (*e.g.*, SPARC's RMO).

Whereas conventional processors enforce ordering constraints conservatively, the vast majority of these ordering stalls are dynamically unnecessary [15]. Hence, researchers have proposed using post-retirement speculation, that is, speculation beyond the instruction window, to eliminate the performance gap between strong consistency models and relaxed consistency models [5, 10, 14, 15, 17, 19, 21, 26, 28, 33].

These proposals take two alternative approaches. One class of proposals directly extends the instruction window with fine-grained buffers for *speculatively retired* instructions, detecting consistency violations by snooping incoming cache coherence requests [14, 15, 26, 28, 33]. This approach has been shown to match or exceed the performance of a conventional RMO implementation. However, tracking speculative state at a per-instruction or per-store granularity requires post-retirement buffers that must grow proportionally to the duration of speculation. Furthermore, these proposals either have rollback or commit cost that is proportional to the duration of speculation. The high store miss latency of current systems can be fully tolerated only by deep speculation, leading to high storage requirements and rollback/commit costs.

A second class of proposals takes a more radical approach by enforcing consistency at coarse granularity on chunks of instructions rather than individual memory operations, thus amortizing the cost of maintaining speculative state and acquiring store permissions [5, 10, 17, 19]. This approach has also been shown to achieve high performance. However, these proposals require unconventional extensions to the memory system, such as efficient support

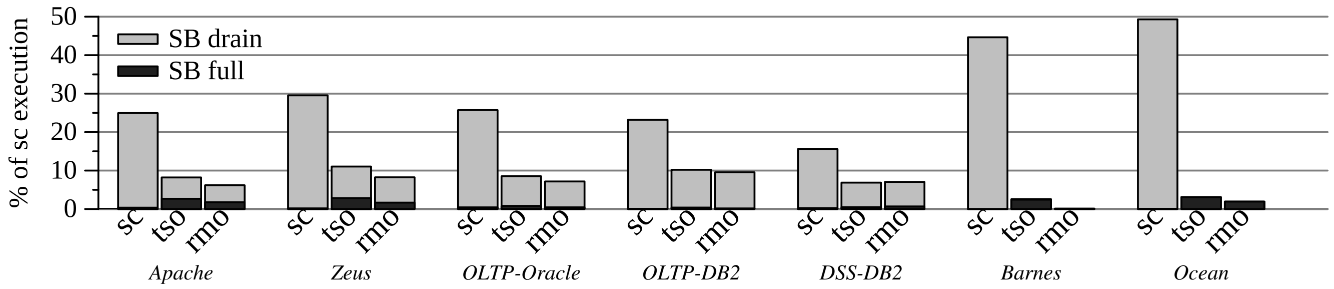

Figure 1. Ordering stalls in conventional implementations of SC, TSO, and RMO as a percent of execution time. The "SB drain" segments represent stall cycles due to store buffer drains triggered by atomic operations and fences (under TSO and RMO) or any memory operation (under SC). The "SB full" segments represent stall cycles due to limited store buffer capacity.

for global commit arbitration, update-based cache coherence protocols, and/or support for bulk operations on read-set and write-set signatures, potentially hindering widespread adoption. Section 2 and Section 5 further describe prior work.

To enable performance-transparent memory ordering in conventional multiprocessors, this work builds upon techniques for deep speculation pioneered in other contexts [2, 11, 16, 18, 20, 23, 24, 25, 27] to create InvisiFence, a new design that uses post-retirement speculation to implement any consistency model efficiently. InvisiFence employs a standard cache coherence protocol, cache hierarchy, and coalescing store buffer sized to hold only outstanding misses (*e.g.*, eight entries). During speculative execution, InvisiFence buffers data for speculative stores in the coalescing store buffer until the miss completes and in the data cache afterwards, using the second level of cache to preserve non-speculative state. InvisiFence detects ordering violations by snooping external cache coherence requests via per-block speculatively read/written bits in the data cache. To abort speculation, InvisiFence flash-invalidates speculatively written blocks and restores checkpointed register state. To commit speculation, InvisiFence simply flash-clears the speculatively read/written bits.

By default, InvisiFence initiates speculation only when the processor would otherwise stall retirement due to consistency constraints (*e.g.*, under SC, when a load cannot retire due to an outstanding store). This *selective speculation* minimizes time spent speculating and consequently vulnerability to rollback-inducing violations. Moreover, InvisiFence opportunistically commits speculation in constant time whenever the ordering requirements for all in-flight memory operations have been satisfied. This instantaneous *opportunistic commit* obviates prior proposals' need to tolerate long-latency commit operations, allowing InvisiFence to obtain high performance with hardware support for only a single in-flight speculation (*i.e.*, only one register checkpoint and one set of bits in the data cache for tracking speculative accesses).

Alternatively, InvisiFence can employ the *continuous speculation* espoused by prior work on chunk-based designs [5, 10, 17, 19]. Instead of initiating speculation only upon a potential ordering stall, continuous speculation executes *all* memory operations speculatively, allowing it to subsume in-window mechanisms for enforcing memory consistency at the cost of a second checkpoint to pipeline chunk commit with subsequent execution. Continuous speculation increases vulnerability to ordering violations, causing a straight-forward implementation to suffer substantial performance degradation relative to selective speculation. To mitigate this penalty, we propose an alternative policy for resolving potential ordering violations: *commit on violate* (CoV). CoV avoids unnecessary rollbacks by deferring — for a bounded timeout interval — those requests that would otherwise cause a violation. This timeout interval provides an opportunity to commit the speculation instead of immediately aborting.

InvisiFence is the first approach for implementing memory consistency that allows deep post-retirement speculation in the context of a standard cache coherence protocol while avoiding fine-grained post-retirement store buffering. Our performance results show that the selective and continuous variants of InvisiFence outperform a conventional RMO implementation. In its highest-performing configuration, InvisiFence adds only an eight-entry coalescing store buffer, a register checkpoint, and two bits per primary data cache block — approximately 1KB of additional state — to a conventional multiprocessor.

2. Background on Memory Consistency

A multiprocessor's memory consistency model specifies the programmer-visible memory reorderings allowed to different memory locations with respect to loads, stores, atomic operations (*e.g.*, compare-and-swap or atomic increment), and explicit memory ordering fences [1]. There are three general classes of consistency models: Sequential Consistency (SC), which guarantees strict memory ordering (*e.g.*, MIPS); Processor Consistency (PC), which relaxes ordering from stores to subsequent loads (*e.g.*, SPARC TSO and x86); and Release Consistency (RC), which relaxes all ordering except at programmer-specified memory fences (*e.g.*, SPARC RMO, PowerPC, ARM, and Alpha). Specific instantiations of the latter two models vary; for concreteness, this paper uses SPARC's TSO and RMO as representative of typical PC and RC models, respectively.

2.1 Conventional Implementations

We describe canonical SC, TSO, and RMO implementations that will serve as reference points for our performance comparisons. These implementations all leverage an invalidation-based cache coherence protocol and a mechanism for in-window speculative memory reordering, but they differ in their ability to employ a post-retirement store buffer and their handling of atomic operations.

Invalidation-based cache coherence protocol. Today's multiprocessors overwhelmingly use block-granularity invalidation-based cache coherence such as snooping or directory protocols. The key properties of these protocols are that they serialize all writes to the same address and inform the processor when a store miss completes. As described below, the processor then leverages these properties to implement its desired memory consistency model without additional help from the coherence protocol.[1]

[1] Although this general approach is used by Intel, AMD, and Sun, there are exceptions. For example, IBM's Power4 requires fences to circulate its ring-based interconnect before completing [30].

Model	Memory Ordering Relaxations	Store Buffer (SB)		Retirement of:			
		Organization	Granularity	Load	Store	Atomic	Full Fence
SC	None	FIFO	Word (8 bytes)	Drain SB	–	Drain SB	N/A
TSO	Store-to-load	FIFO	Word (8 bytes)	–	–	Drain SB	Drain SB
RMO	All	Unordered	Block (64 bytes)	–	–	Complete store	Drain SB

Figure 2. Memory consistency models: definitions and conventional implementations. An entry of "–" indicates that the consistency model imposes no special requirements on retiring the instruction in consideration.

In-window speculation support. Dynamically scheduled processors use a load queue and store queue to support out-of-order execution of memory operations while enforcing in-window uniprocessor memory dependencies. Multiprocessors can similarly support in-window speculative reordering of memory operations while guaranteeing memory ordering by either snooping the load queue whenever a block is invalidated or evicted from the cache [13, 35] or using pre-retirement filtered load re-execution [4, 13, 29]. Such a mechanism is essential for allowing out-of-order load execution in implementations of strongly ordered models (SC and TSO) and allows for in-window speculative execution of memory fences in RMO implementations. Thus we assume such in-window support as part of the baseline implementations of all memory consistency models.

Implementing SC, TSO, and RMO. SC implementations can employ a word-granularity FIFO store buffer, but as loads must stall at retirement until all prior stores complete, the store buffer's utility is limited. TSO implementations, by contrast, allow loads to retire past outstanding stores. However, the size of the FIFO store buffers employed by these implementations is limited by the need to support age-ordered fully-associative search for bypassing values to subsequent loads. Thus, TSO implementations may incur stalls at store retirement due to the store buffer being full. Furthermore, to satisfy ordering constraints at atomic operations the store buffer must drain by stalling until all prior store misses have completed. RMO implementations typically employ an unordered block-granularity coalescing store buffer and allow stores that hit in the cache to skip the store buffer and retire directly into the data cache. The extra capacity and RAM-based nature of a coalescing store buffer typically eliminate store buffer capacity stalls. However, implementations of RMO must drain the store buffer at explicit memory barriers, and they cannot retire an atomic operation until it obtains write permission to ensure atomicity. Unfortunately, memory fences and atomic operations are not infrequent, as they form the foundation on which locks and lock-free synchronization are built. Figure 2 summarizes the differences between conventional implementations of SC, TSO, and RMO.

2.2 Post-Retirement Speculation

Researchers have proposed *post-retirement speculation* to close the performance gap between strong and weak consistency models [5, 14, 15, 19, 28, 33]. The goal of these proposals is to support deeper speculative memory reordering than possible using only in-window mechanisms. These prior proposals can be classified into two broad lineages of work.

Speculative retirement. The first lineage [14, 15, 28, 33] directly attacks memory ordering stalls in conventional implementations by allowing instructions to *speculatively retire* when they would otherwise stall at retirement waiting for a memory ordering constraint to be satisfied. These proposals maintain the state of speculatively-retired instructions at a fine granularity, enabling precise recovery from misspeculations. They detect such misspeculations by snooping external cache coherence requests similarly to in-window mechanisms for speculative reordering, and commit speculative state once all outstanding store misses have completed. Ranganathan *et al.* [28] first introduced the concept of speculative retirement via an implementation that allows loads and non-memory instructions—but not stores—to speculatively retire into an in-order history buffer. Gniady *et al.* [15] extended this implementation to allow stores to speculatively retire and delegated the task of monitoring external requests to a separate RAM-based structure, enabling a larger history buffer. Gniady and Falsafi [14] reduced the amount of custom storage needed to buffer speculative state by recording the speculative history in the memory hierarchy.

More recently, Wenisch *et al.* [33] proposed *atomic sequence ordering* (ASO), which employs register checkpointing rather than a history buffer. The key property of ASO's design is that it performs *all* forwarding from stores to loads via the L1 cache. ASO thus places all speculative data directly into the L1 cache at retirement regardless of whether the block is present or not. As a result, ASO must extend the L1 cache with per-word valid bits to support correct merging of data. ASO adds per-block speculatively-accessed bits to detect violations. As the L1 cache now contains core-private speculative values, the L2 provides data for external coherence requests. To facilitate commit of speculative state into the L2 cache, ASO employs a FIFO store buffer called the Scalable Store Buffer (SSB). The SSB holds all stores from a speculative sequence in-order (because the SSB does not supply values to loads, its scalability is less restricted than a traditional FIFO store buffer). To commit speculation, the processor drains these speculative stores in order from the SSB into the L2 cache while stalling external requests at the L2. Overall, the key advantages of ASO over the earlier proposals are that (1) SSB storage requirements are proportional to the number of stores rather than the number of instructions in a speculative sequence and (2) ASO does not require a separate structure for detecting violations.

Chunk-based enforcement of consistency. Another lineage of work [5, 7, 10, 19] proposed the idea of enforcing consistency at the granularity of coarse-grained chunks of instructions rather than individual instructions. These approaches execute in continuous speculative chunks, buffering register state via checkpoints and buffering speculative memory state in the L1 cache. Correct recovery from misspeculation is ensured by maintaining non-speculative state in lower levels of the cache hierarchy and invalidating speculatively-written lines from the L1 cache on abort to be refetched on demand. Chunks do not attempt to acquire permissions for individual stores during execution but rather acquire permissions for all stores within a chunk via a single operation at the end of the chunk. After acquiring permissions, the chunk sends its write set to other processors, which use this write set to detect violations. The processor tolerates the latency of this commit process via pipelined chunk execution. These proposals' continuous speculation also makes it unnecessary for them to provide a distinct mechanism for detecting in-window memory consistency violations, as all loads are already executing as part of a speculative chunk.

TCC [19] first introduced the concept of enforcing consistency at a coarse granularity. The original TCC implementation employed a global commit token and an update-based coherence protocol, with chunks broadcasting both addresses and data to all other chunks on acquiring commit permissions via global arbitration for the commit token. A subsequent design [7] employs a distributed arbitration mechanism and an invalidation-based coherence protocol in which chunks send addresses but not data of write sets after committing. More recently, Ceze *et al.* [5] proposed BulkSC, which leverages the Bulk [6] architecture to decouple coarse-grained enforcement of consistency from the cache coherence protocol. Bulk maintains the read- and write-sets of speculative chunks as finite-size conservative representations called *signatures* that are small enough to be communicated to arbiters and other processors.

Discussion. The speculative retirement and chunk-based enforcement approaches differ along three key dimensions: maintenance of speculative state at a per-store versus per-block granularity, acquiring store permissions per-block versus per-chunk, and speculating selectively versus continuously. All of these choices have tradeoffs. Per-store state maintenance enables more precise rollback at a cost of requiring much more speculative state: storage requirements grow proportionally to the number of speculatively-retired stores, leading to substantial storage costs (*e.g.*, ASO's SSB as proposed is 10 KB). Chunk-based designs require efficient mechanisms for global arbitration and efficient mechanisms for communication of chunk write sets; although innovative solutions to these problems have been proposed, they depart significantly from conventional memory systems. Finally, selective speculation minimizes the vulnerability to misspeculations, whereas continuous speculation simplifies processor design by decoupling consistency from the processor core.

The next two sections present INVISIFENCE, a consistency model implementation based on post-retirement speculation that leverages designs for deep speculation proposed in other contexts to support both continuous and selective speculation while avoiding per-store buffers and operating within a standard cache coherence protocol.

3. INVISIFENCE Mechanisms

This section describes the structures and operations of INVISIFENCE's post-retirement speculation mechanism. The next section (Section 4) describes INVISIFENCE's use of this mechanism in both a selective speculation mode, which tailors speculation to the requirements of various consistency models, and a continuous speculation mode, which is suitable for any consistency model. Section 5 compares INVISIFENCE to other recent proposals for speculative implementations of memory consistency.

INVISIFENCE uses post-retirement speculation to reduce the performance penalty of atomic operations, memory ordering fences, and the frequent ordering requirements of stronger models such as TSO and SC. INVISIFENCE's implementation is explicitly designed to avoid requiring any per-instruction tracking structures or unconventional mechanisms for acquiring coherence permissions. In fact, our goal for INVISIFENCE is to require only small modifications to the well-understood baseline RMO design presented in the previous section. To accomplish this goal, INVISIFENCE builds upon techniques and mechanisms from the extensive prior work on supporting deep speculation in contexts such as speculative locking and synchronization [24, 27], transactional memory [2, 20], speculative compiler optimizations [25], checkpointed resource reclamation [9, 22, 23], and speculative multithreading [11, 16, 18]. INVISIFENCE, however tailors these techniques for use in the context of eliminating performance penalties of memory ordering.

3.1 INVISIFENCE Structures

INVISIFENCE uses the structures of the baseline RMO implementation described in Section 2, including its processor, block-granularity non-FIFO store buffer, in-window speculation mechanism, write-back caches, and conventional invalidation-based cache coherence protocol. INVISIFENCE makes the following modifications to this baseline processor's structures:

Register checkpoint. As with any checkpoint/recovery scheme, INVISIFENCE relies on the processor's ability to checkpoint and restore its register state and program counter.

Speculative access bits added to the data cache tags. INVISIFENCE adds *speculatively-read* and *speculatively-written* bits to each cache tag entry of the primary data cache. For a 64KB cache with 64-byte blocks, this requires 2k bits (256 bytes), representing 0.4% overhead. INVISIFENCE's read and written bits support two single-cycle flash-clear operations: first, a flash clear of all speculatively-read and speculatively-written bits, and second, a flash conditional-invalidation operation that clears the valid bit of any block that has the speculatively-written bit set. Figure 3 illustrates standard 6T SRAM cells augmented to support these operations. INVISIFENCE uses these operations to provide fast speculation commit and abort (described below).

Store buffer extended with flash invalidation. INVISIFENCE employs a coalescing unordered store buffer sized proportionally to the number of outstanding store misses (*e.g.*, eight block-sized entries). Similar to that of the baseline RMO processor, this store buffer (1) holds retired but not-yet-committed writes, (2) has per-byte valid bits, (3) is not searched by incoming coherence requests, and (4) never provides data to other processors. INVISIFENCE adds the ability to flash-invalidate all speculative entries in the store buffer, used during abort. To avoid incorrectly invalidating non-speculative data, the store buffer does not perform coalescing between speculative and non-speculative stores for a given block.

Optional support for second checkpoint. INVISIFENCE can optionally support a second checkpoint. To do so, INVISIFENCE adds a second register checkpoint and pair of speculative access bits. To avoid having multiple speculative values for a given block in the L1 cache, stores from the second checkpoint to blocks that have also been written by the first checkpoint are kept in the store buffer until the first checkpoint commits.

INVISIFENCE makes no modifications to the primary cache data array, secondary caches, or the coherence protocol.

3.2 INVISIFENCE Operation

We now describe the operations that INVISIFENCE employs to support post-retirement speculation, including initiation of speculation, handling of speculative loads, stores, atomic operations and memory fences, commit of speculation, and detection/recovery from violations.

Speculation initiation. INVISIFENCE initiates speculation by taking a register checkpoint.

Speculative loads. Loads that occur during speculation set the speculatively-read bit for the given cache line. This bit is set either at execution or at retirement of the load depending on the mode in which INVISIFENCE is operating, as discussed in Section 4.

Speculative stores. During speculative execution INVISIFENCE uses its coalescing store buffer as in the baseline RMO processor: store hits retire directly into the L1 cache and store misses retire into the store buffer until the block is filled, at which time the store is moved from the store buffer into the L1 cache. In both cases INVISIFENCE sets the speculatively-written bit of the block when the cache is updated.

To allow recovery, the processor must prevent the only pre-speculative copy of a block from being overwritten and thus lost.

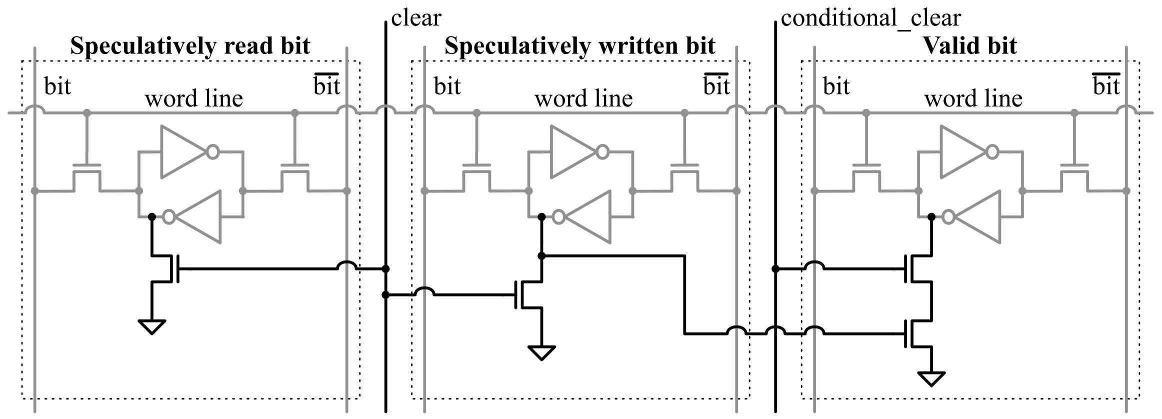

Figure 3. Six-transistor SRAM cells (in gray) augmented with circuitry (in black) for flash-clear (left-most and middle cells) and conditional flash-clear (right-most cell). When the `clear` signal is asserted, both the read and written bits are pulled down to zero. When the `conditional_clear` is asserted, the valid bit is pulled down to zero (invalid) if the speculatively written bit is one.

When a speculative store to a non-speculative dirty block occurs, the processor initiates a clean-writeback of the block to the next level of cache hierarchy, which transitions the block to the non-dirty writable state (Exclusive).[2] The speculative store retires into the store buffer, allowing the cleaning operation to occur in the background without blocking retirement. Once the cleaning operation is complete, the store buffer updates the L1 cache and sets the block's speculatively-written bit.

Speculative atomic operations. An atomic read-modify-write operation (such as an atomic increment) is treated as a pair of normal memory operations, with the restriction that both parts of the atomic operation must be contained within the same speculation to guarantee the atomicity of the read-modify-write operation.

Speculative memory fences. While in speculative execution mode, memory fence operations retire without stalling or waiting for the store buffer to drain.

Speculation commit. To commit speculation, all prior stores must have completed into the cache (*i.e.*, the store buffer must be empty). If the store buffer is not empty, the speculation waits for the store buffer to drain. Once all prior stores have completed, the processor flash-clears the read and written bits in the cache tags to atomically (1) commit all speculative writes and (2) stop tracking speculative reads.

Violation detection. The processor must ensure that the speculative reordering of memory operations never becomes visible to another processor. INVISIFENCE detects potential violations of this invariant by comparing external coherence requests for a block against that block's speculatively-read and speculatively-written bits: an external write request to a speculatively-read block or any external request to a speculatively-written block indicates a potential violation. To ensure detection of all violations, INVISIFENCE prevents speculatively accessed blocks from escaping the cache by forcing a commit before evicting any speculatively-read or speculatively-written block from the data cache (*e.g.*, for capacity or conflict reasons).

The default behavior of INVISIFENCE on detecting a potential violation is to immediately abort speculation. Alternatively, INVISIFENCE may defer the offending incoming request for a bounded timeout interval while it attempts to commit speculation through a policy called *commit on violate*. During this interval, if all the processor's outstanding store misses complete, the processor can commit the speculation. To ensure forward progress, the processor aborts the speculation if it is unable to commit before the timeout interval expires. By giving the speculation an opportunity to commit before resorting to speculation rollback, this policy can substantially reduce the performance penalty of speculation. The experimental evaluation by default assumes the simpler abort-immediately policy, but it also includes experiments with the commit-on-violate policy with a 4000 cycle timeout interval.

Speculation abort. To abort speculation and restore pre-speculative state, the processor flash-invalidates any speculative entries in the store buffer, invokes the conditional-invalidation operation on the cache, and flash-clears the read/written bits. Because the first speculative write to a dirty block always forces a "cleaning" writeback, the pre-speculative value is still available elsewhere in the memory system. These invalidated blocks will thus be restored incrementally on demand via normal cache misses. The processor restarts execution by flushing any in-flight instructions, restoring the register checkpoint, and resuming execution at the program counter. To guarantee forward progress in all cases, the processor completes at least one instruction non-speculatively before initiating any subsequent speculations.

Discussion. INVISIFENCE commits a group of instructions as an atomic unit. At commit, INVISIFENCE's mechanisms enforce the invariant that no speculatively-read value has changed and no other processor has seen a speculatively-written value (because any loss of permissions to a speculatively-accessed block would have triggered an abort). INVISIFENCE additionally ensures that all stores have been written to the L1 cache. By flash-clearing the read/write bits, all stores are made visible to other processors atomically. The entire sequence of speculative memory operations thus commits into the global memory order atomically, allowing operations to be reordered internally without violating consistency invariants [27]. As speculative stores do not escape the first-level data cache, flash-invalidating speculatively-written blocks on abort atomically discards all speculatively-modified versions.

4. INVISIFENCE Speculation Policies

The above mechanisms leave an INVISIFENCE implementation significant freedom in choosing specific policies to determine when to initiate and commit speculation. As discussed in Section 2.2, prior proposals have suggested both speculating selectively and speculating continuously, with tradeoffs to each choice. Inspired by

[2] In the case where there is a non-speculative entry for that block in the store buffer, the non-speculative entry is written into the cache before the cleaning operation is performed.

Variant	Speculates on?	% time speculating?	Min. chunk size?	Snoops load Q?
InvisiFence-Selective$_{rmo}$	Fences, atomics	0-10%	None	Yes
InvisiFence-Selective$_{tso}$	Store/atomic reorderings, fences	10-40%	None	Yes
InvisiFence-Selective$_{sc}$	All memory reorderings	10-50%	None	Yes
InvisiFence-Continuous	Continuous chunks	Near 100%	~100 instructions	No

Figure 4. Properties of InvisiFence variants. "% time speculating" specifies the percentage of time that the variants spend in speculation on our workloads (see Figure 10). "Min. chunk size" is the size that a chunk must be before being allowed to commit.

this previous work, this section presents variants of InvisiFence that support each mode of speculative execution: InvisiFence-Selective speculates only when necessary to minimize risk of violations, and InvisiFence-Continuous speculates continuously to decouple consistency enforcement from the processor. Figure 4 summarizes these proposals.

4.1 InvisiFence-Selective

InvisiFence-Selective initiates speculation only when an instruction would otherwise stall at retirement due to the ordering requirements of the target memory consistency model. Under SC, InvisiFence initiates speculation whenever a load is ready to retire but the store buffer is not empty. Under TSO, InvisiFence initiates speculation when a store or an atomic operation is ready to retire but the store buffer is not empty.[3] Finally, InvisiFence speculates under RMO when either (1) a memory fence is ready to retire but the store buffer is not empty or (2) an atomic operation would stall retirement because of a store miss to the block. Under all models, both register checkpointing and marking of speculatively-read bits for loads are performed at instruction retirement, as it is only at retirement that a given instruction knows whether it is speculative.

InvisiFence-Selective commits speculation opportunistically and in constant-time whenever the store buffer is empty, because an empty store buffer indicates that there are no outstanding store misses and thus that any ordering constraints that induced speculation are now satisfied. At this point, InvisiFence-Selective transitions to non-speculative execution until the next ordering-induced stall. InvisiFence-Selective also commits upon a cache overflow and prior to executing any instruction with irreversible side effects (*e.g.*, memory operations marked as such in the MMU). In such cases, it must wait for the store buffer to drain before committing.

4.2 InvisiFence-Continuous

Based on previous proposals that execute all instructions in speculative chunks [5, 10, 17, 19], InvisiFence-Continuous is a variant of InvisiFence that speculates continuously to subsume in-window mechanisms for enforcing memory consistency. Similar to these previous schemes, in InvisiFence-Continuous loads mark speculatively-read cache bits at execution rather than retirement. As every load is part of some speculative chunk, this policy ensures that any consistency violation will be detected without requiring an in-window mechanism (*e.g.*, load queue snooping). A chunk can commit once all its loads retire and stores complete.

Similar to prior proposals of continuous speculative execution, InvisiFence-Continuous uses more than one in-flight speculation to overlap the commit of a preceding checkpoint with execution of the subsequent checkpoint. To avoid overly-frequent processor checkpointing, InvisiFence-Continuous imposes a minimum chunk size. After a chunk reaches this minimum size a new checkpoint is taken once one is available. Pipelined chunk commit eliminates stalls that would otherwise arise while a chunk is waiting for its memory operations to complete before committing.

5. Comparison to BulkSC and ASO

Of the prior proposals discussed earlier in Section 2, ASO [33] and BulkSC [5] are the two most recent of the lineages of speculative retirement and chunk-based enforcement, respectively. This section differentiates InvisiFence from these two prior proposals along four dimensions: mechanisms for maintaining speculative memory state, mechanisms for acquiring permissions for speculative stores, the commit process, and whether speculation is continuous or selective. Figure 5 summarizes the various proposals' design choices and their implications.

Mechanism for maintaining speculative memory state. InvisiFence, BulkSC, and ASO all maintain speculative state in the data cache for forwarding to subsequent loads. However, they differ in their mechanism for buffering speculative store state. As discussed in Section 2.2, ASO maintains the state of all speculative stores per-store in the Scalable Store Buffer (SSB). Furthermore, it requires per-word valid bits in the L1 cache to enable correct store-to-load forwarding from pending store misses, as the L1 cache rather than the SSB is responsible for forwarding from such pending misses. In contrast, BulkSC and InvisiFence buffer pending store misses in an unordered store buffer and completed stores in the L1 cache at a per-memory-block granularity, requiring less than 1KB of storage for the store buffer and obviating the need for per-word valid bits in the L1 cache.

Mechanism for acquiring permissions for speculative stores. InvisiFence and ASO acquire store permissions eagerly (*i.e.*, as stores are encountered) via a conventional invalidation-based cache coherence protocol. In contrast, BulkSC uses signature-based global arbitration to obtain all write permissions for a chunk via a single operation performed lazily at the time of commit.

The commit process. The above distinctions lead to significantly different speculation commit processes, with implications on commit latency. To tolerate the global arbitration latency involved in its commit process, BulkSC supports multiple in-flight speculative chunks to overlap commit with subsequent execution. In contrast, commit is a local operation in InvisiFence and ASO. Under ASO, however, commit requires draining store values from the FIFO store buffer into the L2 cache. To ensure atomicity, the cache's external interface must be disabled during this process, delaying other coherence activity. As with BulkSC, ASO supports multiple in-flight speculations to hide this commit latency. InvisiFence-Selective's constant-time local commit mechanism and opportunistic commit combine for a constant-time commit process. Hence, InvisiFence-Selective employs only a single checkpoint.

[3] Note that it would be possible to speculate less frequently under TSO by combining InvisiFence with a non-speculative FIFO store buffer. We leave exploration of such a design to future work.

	BulkSC [5]	InvisiFence-Continuous	InvisiFence-Selective	ASO [33]
Speculative execution	Continuous		Selective	
Violation detection	Lazy	Eager		
Preserving memory state	Write back dirty blocks			Stores write-thru to L2
Commit mechanism	Global arbitration	Flash-clear read/written bits		Drain stores from SSB to L2
Commit latency	Grows with # of processors	Constant-time		Grows with chunk size
Requires multiple checkpoints?	Yes		No	Yes
Fwding from unfilled blocks	Coalescing store buffer			L1 cache
Impact on memory system	Global transfer of signatures	Read/written bits in L1 cache		Read/written, sub-block bits
Avoids load queue snooping?	Yes		No	

Figure 5. Comparison of speculative implementations of memory consistency.

Processing Nodes	UltraSPARC III ISA 4 GHz 8-stage pipeline; 4-wide out-of-order 96-entry ROB, LSQ
Store Buffer	SC, TSO: 8-byte 64-entry FIFO RMO, InvisiFence: 64-byte 8-entry coalescing InvisiFence-Continuous: 64-byte 32-entry
L1 Caches	Split I/D, 64KB 2-way, 2-cycle load-to-use 3 ports, 32 MSHRs, 16-entry victim cache
L2 Cache	Unified, 8MB 8-way, 25-cycle hit latency 1 port, 32 MSHRs
Main Memory	3 GB total memory, 40 ns access latency 64 banks per node, 64-byte cache blocks
Protocol Controller	1 GHz microcoded controller 64 transaction contexts
Interconnect	4x4 2D torus, 25 ns latency per hop 128 GB/s peak bisection bandwidth

Figure 6. Simulator parameters.

Continuous versus selective speculation. As discussed in Section 2, selective speculation (used by ASO and InvisiFence-Selective) reduces the window of vulnerability to violations, whereas continuous speculation (used by BulkSC and InvisiFence-Continuous) unifies the in-window and post-retirement detection of ordering violations, thus eliminating the need for a distinct in-window mechanisms for enforcing memory ordering.

Summary. Although InvisiFence shares attributes with ASO and BulkSC, InvisiFence is the first proposal to implement memory consistency via post-retirement speculation without requiring either fine-grained buffers to hold speculative state or requiring global arbitration for commit of speculation. By maintaining state for pending speculative stores in an unordered coalescing store buffer and state for completed speculative stores in the L1 cache, InvisiFence avoids ASO's large SSB and its sub-block valid bits on L1 cache blocks (reducing dedicated storage requirements by a factor of 15). InvisiFence's store buffer capacity requirement is independent of speculation depth. Instead, it depends only on the number of simultaneous store misses. By leveraging a conventional invalidation-based cache coherence protocol to acquire store permissions and detect violations, InvisiFence avoids BulkSC's global arbitration for chunk commit permissions and use of non-standard chunk-based communication mechanisms. Finally, InvisiFence supports both selective and continuous speculation in the context of a standard cache coherence protocol.

Web Server	
Apache	16K connections, fastCGI, worker threading model
Zeus	16K connections, fastCGI
Online Transaction Processing (TPC-C)	
OLTP-DB2	100 warehouses (10 GB), 64 clients, 450 MB buffer pool
OLTP-Oracle	100 warehouses (10 GB), 16 clients, 1.4 GB SGA
Decision Support (TPC-H on DB2)	
DSS-DB2	Query 2, 450 MB buffer pool
Scientific	
Barnes	16K bodies, 2.0 subdiv. tol.
Ocean	1026x1026 grid, 9600s relaxations, 20K res., err tol 1e-07

Figure 7. Workloads.

6. Experimental Evaluation

Our evaluation demonstrates experimentally that InvisiFence effectively eliminates ordering penalties, providing a performance-transparent implementation of memory ordering in conventional multiprocessors. Furthermore, we investigate different InvisiFence policies, highlighting the ways in which consistency model variations affect performance.

6.1 Methodology

We model InvisiFence using the Flexus 3.0.0 [34] full-system multiprocessor simulation infrastructure. Flexus extends Virtutech Simics' SPARC v9 functional model with detailed models of an out-of-order processor core, cache hierarchy, protocol controllers and interconnect. We study InvisiFence in the context of a 16-core directory-based shared-memory multiprocessor. We configure Flexus to approximate the Intel Core 2 microarchitecture. Figure 6 provides the configuration details of our baseline system model.

We performed sensitivity studies (not shown) to determine store buffer capacities for InvisiFence that provide performance close to that of a store buffer of unbounded capacity. For InvisiFence configurations that employ a single checkpoint, a store buffer with eight entries suffices. Configurations of InvisiFence that employ two in-flight checkpoints (which includes InvisiFence-Continuous) use a 32-entry store buffer; this larger store buffer compensates for the increased pressure caused by keeping stores from the second checkpoint in the store buffer until the first checkpoint commits if those stores are to blocks previously written by the first checkpoint.

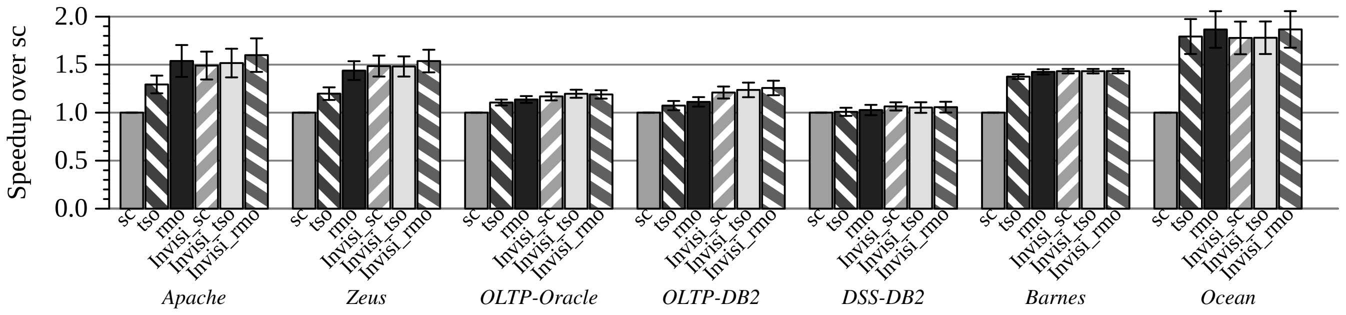

Figure 8. Speedups of InvisiFence over conventional consistency model implementations.

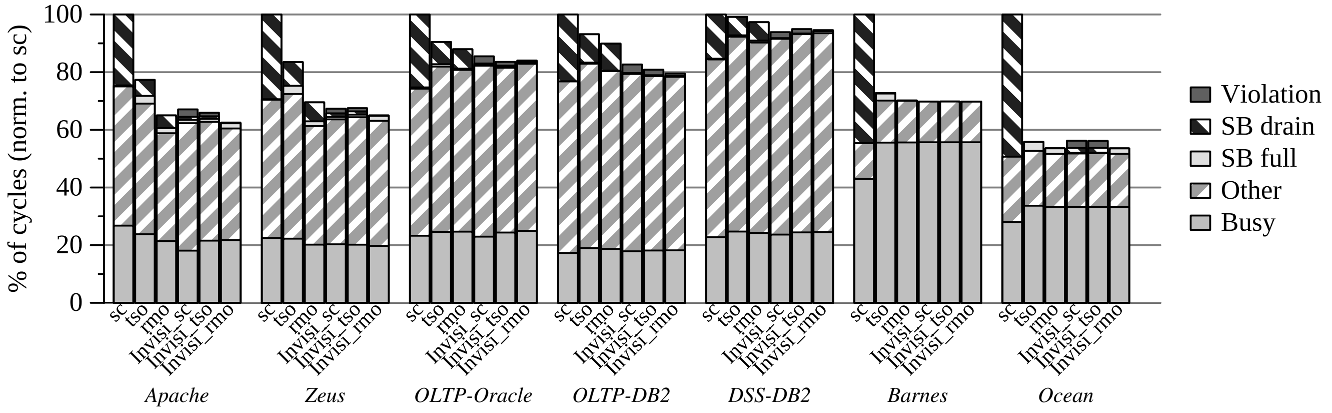

Figure 9. Time breakdown of InvisiFence and conventional consistency model implementations.

Figure 7 describes the set of commercial and scientific applications we use to evaluate InvisiFence. We measure performance using the SimFlex multiprocessor sampling methodology [34] and report 95% confidence intervals on speedup results. Our samples are drawn over an interval of 10s to 30s of simulated time for OLTP and web server applications, over the complete query execution for DSS, and over a single iteration for scientific applications.

Flexus models both the functional and performance impact of execution under conventional and speculative implementations of the SC, TSO, and RMO consistency models. Flexus performs in-window speculative load execution and store prefetching. We extend Flexus' existing post-retirement speculative consistency support with an implementation of InvisiFence. Section 6.4 compares the performance of InvisiFence-Selective against the ASOsc [33] post-retirement speculation implementation available in the public Flexus release.

Several of the commercial applications we study require TSO or stronger consistency for correct execution. We use the same methodology as prior work [33] to approximate the execution of these workloads under RMO by inserting memory fences at all lock acquires. This methodology is unable to introduce fences at lock release because it is difficult to reliably identify the releasing store for the complex lock implementations used by our workloads. Hence, this model strictly overestimates the performance of conventional RMO, conservatively underestimating the performance benefits of post-retirement speculation. Similar to previous work [33], our simulator separately tracks TSO-consistent execution and rolls back on a mismatch; these rollbacks are extremely rare and have negligible performance impact.

6.2 Conventional Implementations

As foreshadowed by Figure 1 and demonstrated by prior work [5, 12, 15, 26, 28, 32, 33], Figure 8 shows that varying ordering constraints introduce substantial penalties in conventional memory consistency implementations (*i.e.*, those described in Section 2.1). The three left-most bars in each group in Figure 8 show the relative performance of conventional SC, TSO, and RMO implementations (higher is better) for our workloads. The FIFO store buffer enabled by TSO's relaxation of store-to-load ordering allows TSO to outperform SC by 24% on average. RMO's further relaxations provide little advantage over TSO for some workloads (*e.g.*, Barnes and Ocean), but provide significant benefit in other workloads (*e.g.*, Apache and Zeus). On average, RMO outperforms TSO by 8% for these workloads.

Figure 9 plots normalized runtimes (the inverse of the speedups of Figure 8) and divides the execution runtime into various components (on this graph, lower is better). The five runtime components are: "Busy" (cycles actively retiring instructions), "Other" (stall cycles unrelated to memory ordering, *e.g.*, load misses), "SB full" (cycles that a store is stalling retirement waiting for a free store buffer entry), "SB drain" (cycles stalling until the store buffer drains because of an ordering requirement, *e.g.*, for a fence in a conventional RMO implementation), and "Violation" (cycles spent executing post-retirement speculation that ultimately rolls back due to a violation of memory ordering).

Although relaxing memory ordering constraints can improve performance substantially, the execution time breakdown in Figure 9 shows that conventional implementations of relaxed consistency are not sufficient to avoid all performance penalties from memory ordering enforcement. Under TSO, substantial "SB full"

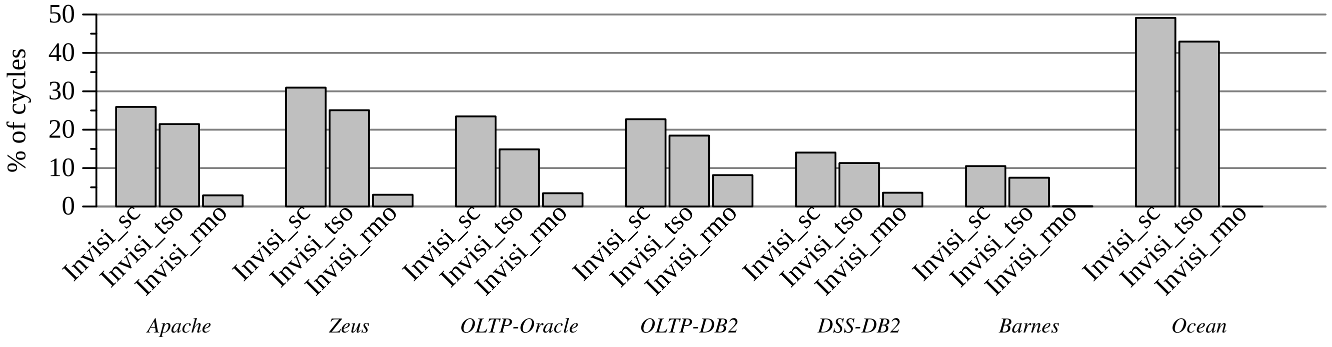

Figure 10. Percent of cycles that INVISIFENCE variants spend in speculation.

stall cycles occur because a FIFO store buffer does not scale to the capacity required during store bursts. TSO also suffers from "SB drain" stalls because of atomic operations. RMO mostly avoids "SB full" cycles, but memory ordering fences induce "SB drain" stalls. Although RMO incurs no memory ordering stalls in the two scientific workloads (Barnes and Ocean), the frequent synchronization in the other workloads prevents the conventional implementation of RMO from hiding all memory ordering penalties.

6.3 INVISIFENCE-SELECTIVE

The three right-most bars of each group in Figure 8 and Figure 9 represent the corresponding variants of INVISIFENCE-SELECTIVE configured to enforce SC, TSO, and RMO (labeled Invisi_sc, Invisi_tso, and Invisi_rmo respectively). As these graphs indicate, even the strictest variant of INVISIFENCE-SELECTIVE (INVISIFENCE$_{sc}$) outperforms the conventional implementations of SC, TSO, and RMO by 36%, 9% and 2% respectively. The runtime breakdown in Figure 9 shows that INVISIFENCE$_{sc}$'s performance improvement over these conventional implementations arises primarily from a substantial reduction in memory ordering penalty cycles ("SB full" and "SB drain").

When using INVISIFENCE-SELECTIVE to enforce weaker ordering constraints, INVISIFENCE achieves even higher performance. As Figure 9 shows, INVISIFENCE$_{rmo}$ eliminates virtually all stalls related to memory ordering. Consequently, INVISIFENCE$_{rmo}$ outperforms the conventional implementation of RMO by as much as 13% and by 5% on average, demonstrating that speculation has a beneficial effect even for relaxed consistency models. INVISIFENCE$_{rmo}$ also outperforms INVISIFENCE$_{sc}$ and INVISIFENCE$_{tso}$ by as much as 7% and 5% respectively (3% and 2% on average). These performance gains demonstrate the advantage of executing software under the most relaxed memory consistency model it supports.

As illustrated by the reduction in "Violation" cycles in Figure 9 for INVISIFENCE$_{rmo}$ versus INVISIFENCE$_{sc}$, the weaker memory models incur fewer wasted cycles due to aborted speculations caused by potential ordering violations. The impact of violations decreases primarily because the weaker models spend fewer cycles executing speculatively. Figure 10 shows the percent of cycles spent in speculation for INVISIFENCE-SELECTIVE for SC, TSO, and RMO. Whereas INVISIFENCE$_{rmo}$ spends less than 10% of time in speculative execution, INVISIFENCE$_{sc}$ and INVISIFENCE$_{tso}$ spend up to 50% of cycles speculating.

6.4 Experimental Comparison to ASO

ASO is a closely-related prior proposal, and prior work has shown ASO has a similar ability to eliminate memory ordering penalties. We have already discussed how INVISIFENCE-SELECTIVE addresses the significant implementation challenges of ASO (Section 5), so the focus of this section is to show that INVISIFENCE-SELECTIVE and ASO achieve similar performance. Comparing the two left-most bars of each group in Figure 11 (lower is better) shows that both ASO and INVISIFENCE-SELECTIVE eliminate almost all memory ordering stalls (as indicated by the small size of the "Violation", "SB drain" and "SB full" segments). Correspondingly, they have similar runtime, with ASO slightly outperforming INVISIFENCE (by 1% on average and at most 5%) due to less time spent performing speculative work that is later discarded due to a violation.

Upon further investigation, we found ASO's use of multiple in-flight speculations is mostly responsible for this small performance difference, because ASO periodically takes checkpoints during speculative execution to reduce the amount of work discarded when violations occur. Adding a second in-flight speculation to INVISIFENCE-SELECTIVE can close the performance gap. The right-most bar of each group in Figure 11 shows that when we modified INVISIFENCE-SELECTIVE to exploit two in-flight speculations, the performance gap between it and ASO disappears (the difference in average performance is negligible). However, as the performance penalty of eschewing multiple checkpoints is only 1% on average, the additional design and verification complexity of supporting multiple checkpoints in INVISIFENCE-SELECTIVE is likely not justified.

6.5 INVISIFENCE-CONTINUOUS

By adopting the continuous speculation approach of prior work [5, 19], INVISIFENCE-CONTINUOUS inherits the ability to eliminate the need for a separate conventional mechanism for detecting in-window memory reordering violations (*e.g.*, load queue snooping). Figure 12 shows the runtime (lower is better) of INVISIFENCE-CONTINUOUS using the abort-immediately policy (labeled Invisi_cont) as compared to conventional SC, RMO, and INVISIFENCE$_{rmo}$. On average, INVISIFENCE-CONTINUOUS achieves a 27% speedup over conventional SC. However, INVISIFENCE-CONTINUOUS does not perform as well as either conventional RMO or INVISIFENCE$_{rmo}$ (which outperform INVISIFENCE-CONTINUOUS by an average of 5% and 10%, respectively). Furthermore, in two cases the performance of INVISIFENCE-CONTINUOUS falls behind that of conventional SC.

The cause of this performance degradation is INVISIFENCE-CONTINUOUS's significant "Violation" cycles on these workloads. Because INVISIFENCE-CONTINUOUS spends essentially all of execution time in speculation, it is significantly more vulnerable to violations than INVISIFENCE-SELECTIVE. Detailed investigations of this effect (results omitted for brevity) indicate that the in-

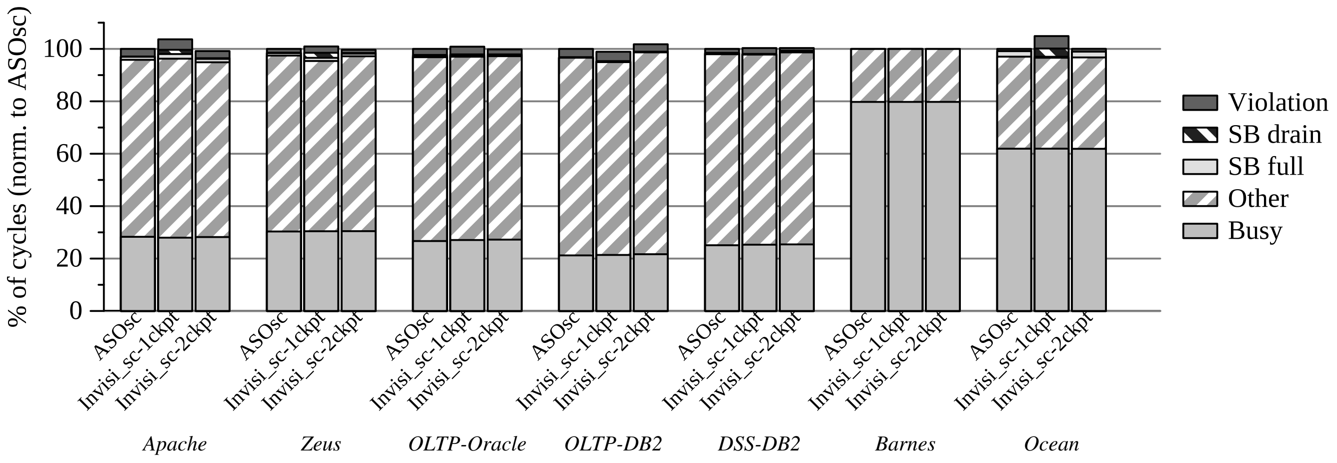

Figure 11. Runtime of ASO, InvisiFence, and InvisiFence with two checkpoints.

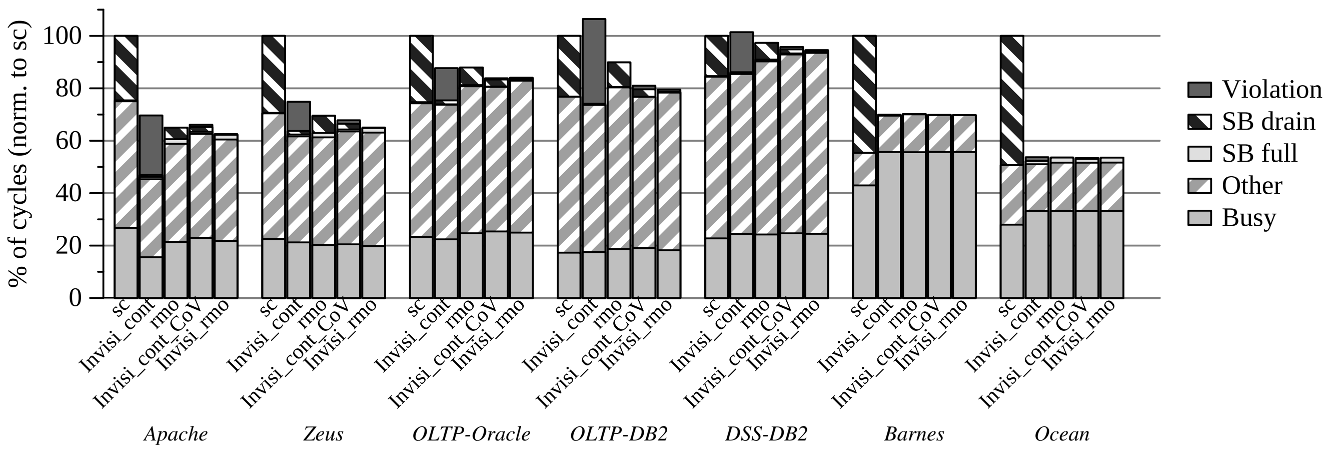

Figure 12. Runtime of SC, InvisiFence-Continuous, RMO, InvisiFence-Continuous$_{CoV}$, and InvisiFence$_{rmo}$.

creased violations of InvisiFence-Continuous do in fact arise because of this increased time in speculation (as opposed to the other minor behavioral and hardware differences between InvisiFence-Continuous and InvisiFence-Selective, *e.g.*, marking speculatively-accessed bits in the cache at load execution rather than retirement).

6.6 Impact of the Commit-on-Violate Policy

Employing the commit-on-violation (CoV) policy (Section 4.2) in InvisiFence-Continuous substantially improves its performance. As indicated by the fourth bar in Figure 12, InvisiFence-Continuous$_{CoV}$ nearly eliminates the lost cycles due to aborts caused by memory ordering violations. This reduction in violations has a first-order effect on performance: using CoV increases performance by as much as 31% and by 8% on average. InvisiFence-Continuous$_{CoV}$ outperforms conventional RMO by an average of 3% and provides most of the performance benefits of InvisiFence-Selective (InvisiFence$_{rmo}$ is on average only 2% faster than InvisiFence-Continuous$_{CoV}$).

We have also investigated CoV in the context of InvisiFence-Selective (results not shown), but as InvisiFence-Selective has far fewer aborts than InvisiFence-Continuous, the performance benefits of CoV are negligible (less than 1% on average).

7. Conclusions

We have presented InvisiFence, a new design for speculative memory consistency that enables performance-transparent memory ordering in conventional multiprocessors under any consistency model. InvisiFence is based on well-understood post-retirement speculation mechanisms proposed in other contexts. By choosing appropriate policies for when to initiate speculation, InvisiFence can employ *selective speculation* to exploit the underlying system memory model to reduce vulnerability to rollback or *continuous speculation* to subsume in-window memory ordering speculation mechanisms. In its highest-performing configuration, InvisiFence requires only a single register checkpoint, two bits per L1 cache block, and an eight-entry coalescing store buffer — less than 1KB of additional state over a conventional multiprocessor.

InvisiFence joins the growing body of work (*e.g.*, checkpointed early load retirement, speculative compiler optimizations, speculative locking, and best-effort transactional memory) that exploits similar, simple post-retirement speculation mechanisms. InvisiFence uses such mechanisms to provide an avenue to substantially improve the performance of existing software, whether written for strict or relaxed consistency models.

Acknowledgments

The authors thank Christopher Batten, Adam Butts, Luis Ceze, Babak Falsafi, Santosh Nagarakatte, Arun Raghavan, and Amir Roth for comments on this work. This work was supported in part by donations from Intel Corporation and National Science Foundation awards CCF-0541292, CCF-0644197 and CCF-0845157.

References

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

[2] L. Baugh, N. Neelakantam, and C. Zilles. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 115–126, June 2008.

[3] R. Bhargava and L. K. John. Issues in the Design of Store Buffers in Dynamically Scheduled Processors. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–87, 2000.

[4] H. Cain and M. Lipasti. Memory Ordering: A Value-Based Approach. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

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

[6] L. Ceze, J. M. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, June 2006.

[7] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, Feb. 2007.

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

[9] A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez. Toward Kilo-Instruction Processors. *ACM Transactions on Architecture and Code Optimization*, 1(4), Dec. 2004.

[10] M. Galluzzi, E. Vallejo, A. Cristal, F. Vallejo, R. Beivide, P. Stenström, J. E. Smith, and M. Valero. Implicit Transactional Memory in Kilo-Instruction Multiprocessors. In *Asia-Pacific Computer Systems Architecture Conference*, pages 339–353, 2007.

[11] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors. In *Proceedings of the Ninth Symposium on High-Performance Computer Architecture*, Feb. 2003.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared Memory Multiprocessors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, Apr. 1991.

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

[14] C. Gniady and B. Falsafi. Speculative Sequential Consistency with Little Custom Storage. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 179–188, Sept. 2002.

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

[16] S. Gopal, T. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th Symposium on High-Performance Computer Architecture*, Feb. 1998.

[17] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with Transactional Coherence and Consistency (TCC). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, Oct. 2004.

[18] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, Oct. 1998.

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

[20] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

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

[22] M. Kirman, N. Kirman, and J. F. Martinez. Cherry-MP: Correctly Integrating Checkpointed Early Resource Recycling in Chip Multiprocessors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2005.

[23] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.

[24] J. F. Martinez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[25] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 174–185, June 2007.

[26] 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 International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, Oct. 1996.

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

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

[29] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 458–468, June 2005.

[30] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

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

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

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

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

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