Non-Speculative Load-Load Reordering in TSO

Alberto Ros[1] Trevor E. Carlson[2] Mehdi Alipour[2] Stefanos Kaxiras[2]
[1] Department of Computer Engineering [2] Department of Information Technology
University of Murcia, Spain Uppsala University, Sweden
aros@ditec.um.es;{trevor.carlson,mehdi.alipour,stefanos.kaxiras}@it.uu.se

ABSTRACT

In Total Store Order memory consistency (TSO), loads can be speculatively reordered to improve performance. If a load-load reordering is seen by other cores, speculative loads must be squashed and re-executed. In architectures with an unordered interconnection network and directory coherence, this has been the established view for decades. We show, for the first time, that it is not necessary to squash and re-execute speculatively reordered loads in TSO when their reordering is seen. Instead, the reordering can be hidden form other cores by the coherence protocol. The implication is that we can irrevocably bind speculative loads. This allows us to commit reordered loads out-of-order without having to wait (for the loads to become non-speculative) or without having to checkpoint committed state (and rollback if needed), just to ensure correctness in the rare case of some core seeing the reordering. We show that by exposing a reordering to the coherence layer and by appropriately modifying a typical directory protocol we can successfully hide load-load reordering without perceptible performance cost and without deadlock. Our solution is cost-effective and increases the performance of out-of-order commit by a sizable margin, compared to the base case where memory operations are not allowed to commit if the consistency model could be violated.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**;

KEYWORDS

Cache coherence, memory consistency, TSO, load reordering, out-of-order commit

ACM Reference format:
Alberto Ros, Trevor E. Carlson, Mehdi Alipour and Stefanos Kaxiras. 2017. Non-Speculative Load-Load Reordering in TSO. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
http://dx.doi.org/10.1145/3079856.3080220

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
http://dx.doi.org/10.1145/3079856.3080220

1 INTRODUCTION

To maximize performance, dynamically-scheduled superscalar processors execute instructions out-of-order and can speculatively transgress the ordering rules of a memory consistency model. *If caught*, they squash incorrect execution and return to a prior checkpointed state that does not violate the desired consistency model. This ability is built-in: checkpointing-and-rollback facilities are already provided to support speculative execution past predicted branches or past unknown memory dependencies.

Example: Total Store Order (TSO) requires a replay of speculative loads that violate load→load program order when a reordering is "detected" by other cores. Load-load reordering can occur, for example, when an older load has an unresolved address or if it misses in the cache while a younger load hits (i.e., a *hit-under-miss*). The younger load is speculative until the older load completes. A coherence event such as invalidation for the address of the speculative load means that the reordering has been "seen" by a core racing with the load in question. In this case, the speculative load and all its dependent instructions, which followed it in execution, are typically squashed and must be reissued.

Motivation: Ideally, we would like a solution in which a speculatively reordered load (and its dependent instructions) would not have to be squashed for consistency enforcement. The benefit, in this case, would be to reduce squash and re-execute overheads, but the opportunity is rare: it occurs only when the speculative reordering is observed.

Nevertheless, such an approach could be invaluable when we *irrevocably bind* reordered loads:

- in stall-on-use, in-order cores that continue executing after a miss without a checkpoint, as for example the DEC Alpha 21164 EV5 that implemented early commit of loads (ECL) [20];
- in decoupled access-execute accelerators where there is a need for non-speculative decoupling via ECL, as for example in DeSC [21];
- in out-of-order cores if we commit out-of-order [5].

Although our solution applies to each of the cases above, we describe it for out-of-order commit which is the most general. For out-of-order commit the known solutions to deal with load-load reordering are:

(1) resort to a weak memory model [5, 20, 37];
(2) *add additional speculation*, i.e., rely on additional speculative state and rollback mechanisms outside the core to undo committed instructions and architectural state [29, 37];
(3) *wait for it*, i.e., refrain from committing a load out-of-order until all previous memory instructions are *performed*

and thus it is determined that the reordering has not been "seen" [5];

(4) *restrict the architecture and wait for it (less)*, i.e., rely on a globally ordered network (e.g., Gigaplane [34]) and refrain from committing a load out-of-order until all previous memory instructions are *ordered* in the network [5, 25, 39].

All four options are problematic. The first restricts the memory model, excluding a vast installed base of machines and software. The second one carries significant cost and complexity and a host of other issues of undoing committed architectural state (e.g., undoing I/O interactions). The third option means that the performance benefit of out-of-order commit is suppressed in the common case to fend off a rare occurrence. Unfortunately, this behavior is encountered not only in parallel programs where we expect consistency enforcement, but even in serial programs, as the processor cannot a-priori guarantee absence of sharing (e.g., coherent I/O). The fourth one is an improvement over the third, provided that we restrict the architecture to globally-ordered busses and snooping coherence [5, 34, 37] or optical networks that provide Atomic Coherence [39]. The improvement is that we only have to wait for previous memory operations to become ordered, not performed. Unfortunately, even with this improvement, there is nothing that can be done for a reordering that involves *unresolved addresses*.

More importantly, as far as we know, there has never been a prior solution which commits (reordered) loads out-of-order (without waiting and without speculating) and can apply to any architecture, including general unordered interconnection networks and directory coherence —the prevailing architectural choices of today. Moreover, even when we rely on ordered networks and snooping coherence, there has never been ***any*** *solution to commit a reordered load over an unresolved address.*

A New Solution: We offer a new perspective on how to reorder loads in TSO without resorting to speculation. We show that it is possible to have *non-speculative* load-load reordering for a general unordered network and directory coherence: we allow irrevocable binding of reordered loads (e.g., out-of-order commit), even when the reordering is over loads with unresolved addresses, and we guarantee at the coherence layer that TSO is preserved.

The main idea of this paper is simple: Consistency is violated when an ordering transgression is seen but cannot be undone. When a memory operation is about to "see" a transgression, the coherence layer *covers up the impending exposé by delaying the operation*, until the transgression disappears without being seen (Section 3). We apply this idea in TSO, where if a store sees a load–load reordering it would immediately make it illegal. In the off-chance this rare event occurs, the coherence protocol steps in and *delays the store in its store buffer* —an action that is inherently allowed in TSO— until the load reordering cannot be seen any longer.

We modify the transactions of a base directory protocol to handle these rare events without penalizing the performance in the common case. Our solution requires negligible additional state at the cores and does not add to the cost of coherence in the caches or the directory. The end result is a deadlock-free and livelock-free solution that yields higher performance for out-of-order commit or higher efficiency for in-order commit processors by not having to squash and re-execute.

Core 0	Core 1
Initially	x=0; y=0;
`ld ra,y`	`st x,1`
`ld rb,x`	`st y,1`

Table 1: TSO does not allow `ra==1` and `rb==0`. If `ld rb,x` hits and irrevocably binds to an old copy of `x` (`x==0`) but `ld ra,y` misses and sees the new value of `y` (`y==1`) we violate TSO.

Why TSO? Total Store Order (e.g., SPARC TSO [36], and TSOx86 [33]) allows for store buffers and relaxes ordering from stores to subsequent loads (relaxes the store→load order, but maintains load→load, store→store, and load→store). TSO implementations actually may reorder far more than what is implied by the formal TSO definition, as long as such reordering is transparent to the programmer. This is easily achieved with speculation and checkpointing mechanisms that are readily available to support instruction-level parallelism (ILP). The example of the speculative load-load reordering in the beginning of this section briefly demonstrates the mechanism.

In this paper, we focus on TSO for two reasons: First, TSO is today one of the most widely used consistency models, as it strikes a good balance between what the programmer expects as reasonable and what the hardware can do for performance [35]. Second, inherent in TSO is an important property: *the ability to delay stores in memory order with respect to loads*. We exploit this property to untangle reordered memory operations at the coherence protocol without the danger of deadlock.

Evaluation and results: We implement an out-of-order commit model in simulation (GEMS with an out-of-order processor model), and with our solution at the coherence layer, we relax the requirement to delay commit for load-load reordering, assuming an architecture with an unordered interconnection network and directory coherence. In the rare case when a store would uncover the reordering, our solution at the coherence layer simply delays the store to preserve TSO.

We quantify the effects of our approach on parallel workloads (SPLASH and PARSEC) and show that: i) we introduce imperceptible overhead with our modifications to the baseline directory protocol; and ii) we achieve performance benefits of 15.4% on average (up to 41.9%) over in-order commit and 10.2% on average (up to 28.3%) over out-of-order-commit without our solution.

2 NOW YOU SEE ME, NOW YOU DON'T

In this section, we explain the idea of non-speculative load–load reordering in TSO. In the discussion that follows (including Section 3) we assume a typical in-order-commit core. We explain how non-speculative reordering can be exploited for out-of-order commit in Section 4.

Assume that out-of-order execution allows the reordering of two loads: the oldest load cannot issue because it has not resolved its address or issues but misses in the cache, while the younger load hits in the cache. We remind here that the younger load would become *speculative* until the time that the older load *is performed*, i.e., resolves its address, issues, and gets its data. In the terminology of Duan, Koufaty and Torrellas the younger load is *M-speculative* [14].

Case	Value y,x	TSO interleaving
①	old, old	`ld y` → `ld x` → `st x` → `st y`
②	old, new	`ld y` → `st x` → `ld x` → `st y`
③	old, new	`ld y` → `st x` → `st y` → `ld x`
④	old, new	`st x` → `ld y` → `st y` → `ld x`
⑤	old, new	`st x` → `ld y` → `ld x` → `st y`
⑥	new, new	`st x` → `st y` → `ld y` → `ld x`
⑦	new, old	`ld x` → `st x` → `st y` → `ld y`
		illegal: `ld y` cycles to `ld x`

Table 2: ①–⑥: the six possible *legal* TSO interleavings for the example in Table 1. TSO is violated only when `ld y` binds to the new value and `ld x` to the old. This corresponds to the *illegal* interleaving ⑦.

For simplicity, we will use the hit-under-miss example (also used in Table 1 in the introduction) to explain the basic mechanism.

An invalidation for the address of the younger M-speculative load sees the reordering. This results in the squash of the load (and all following instructions) and its eventual re-issue, wasting both energy and bandwidth. Assuming that we cannot squash the younger load (for example, because we committed it out-of-order), *we cannot allow anyone to see the reordering*.

Reordering that can violate TSO: The only way a reordering can be observed is when the two reordered loads obtain their values in a different order than the memory order the values were written to the respective memory locations. Both memory locations must change in a certain order for the load reordering to matter. This implies a *happens-before* relation between the respective stores that change the memory locations.

Consider the code in Table 1 where one core reads (loads) and another writes (stores) the same two variables but in the opposite order. Table 2 gives six legal TSO interleavings (①–⑥) that preserve the program order of the loads and the stores. The legal interleavings allow only three combinations of values to be loaded by loads `ld y` and `ld x`, respectively: {*old*, *old*}, {*old*, *new*}, {*new*, *new*}, where *old* is the value before the write (e.g., 0) and *new* is the value after the write (e.g., 1).

Swapping the loads in ①, ⑤, and ⑥ has no effect as it yields the same result: {*old*, *old*}, {*old*, *new*}, and {*new*, *new*} respectively. Swapping the loads in ②, and ④ also yields valid results, {*old*, *old*} and {*new*, *new*} respectively —albeit different from the initial interleaving. This means that a reordering of the loads in these five interleavings *does not matter*. Let us see now what happens if we swap the loads in interleaving ③.

Consider what would happen if the younger load, `ld x`, hits in the cache and binds to the old value and the older load, `ld y`, misses in the cache and sees the new value of y. This implies the interleaving marked ⑦ in Table 2 occurs (i.e., ③ with the loads swapped), which is *illegal* in TSO. Figure 1.A (left diagram) shows why: the program-order between the loads and the program-order between the stores *must* be respected, yet the values read by the loads imply an interleaving that forms a *cycle*. The reason for the cycle is apparent on the right side of Figure 1.A where we show how time flows and how the program-order between loads is violated.

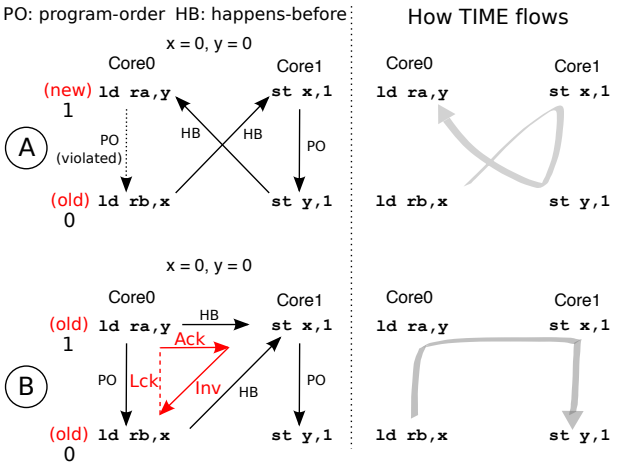

Figure 1: (a) Interleaving ⑦ implies that `ld x` must see an invalidation from `st x` before `ld y` sees the new value from `st y`. (b) Delaying the `Ack` of the invalidation, `Inv`, via a "lockdown," `Lck`, forces `ld y` to *happen-before* `st x` and consequently `st y`.

If we irrevocably bind `ld x` (e.g., if we cannot squash it because we have committed it out-of-order), our only choice to maintain TSO, according to Table 2, is for `ld y` to also bind to the *old value* of y. If `ld y` sees the new value of y, written by core 1, we violate TSO.

Observe now that the necessary condition for `ld y` to see the *new* value of y is that `st x` must be performed: `st x` precedes `st y` in core 1, therefore `st x` must be performed in the system (globally visible) before y gets its new value. This is the key property that we exploit in our approach: As long as we can guarantee that `ld y` will read y *before the store of* x *is performed* we guarantee that `ld y` will get the *old value* of y.

Furthermore, the necessary condition for `ld x` to read the old value of x is for core 0 to have a cached copy of x created before `st x`.[1] This means that core 0 must see an invalidation for x *before* `ld y` can see the new value of y. This gives us the mechanism to delay `st x`.

More specifically, when we get the invalidation for x, we delay its acknowledgement, and therefore we delay `st x` by withholding its write permission, *until* `ld y` gets the old value. This is illustrated in Figure 1.B (left diagram), where the acknowledgment (`Ack`) to the invalidation (`Inv`) of `st x` is delayed with a *lockdown* (`Lck`) of `ld x` until `ld y` performs. The untangling of the reordering in *time* is shown in Figure 1.B in the diagram on the right. Effectively, both loads happen before both stores (interleaving ⑦ turns into interleaving ① in Table 2) and their reordering does not matter. We explain the *lockdown* in the next section.

There are four important observations to be made here:

First, what we do is perfectly legal: protocol correctness cannot depend on the latency of the response to an invalidation, as long as we guarantee that we respond to it. We delay the invalidation response until `ld y` is performed.[2]

[1] It does not matter if this copy is created by `ld x` or if it preexisted.

[2] Worst case delay is when `ld x` waits for a number of *dependent* loads to resolve. The number of dependent loads can be up to one less than the size of the load queue (LQ).

Core 0	Core 1	Core 2
Initially x=0;	y=0;	x cached in Core 0
`ld ra,y`	`st x,1`	`while(rc==0) ld rc,x`
`ld rb,x`		`st y,1`

Table 3: `st x` and `st y` are ordered by a happens-before relationship. `ld x` commits out-of-order and delays the invalidation of `x` until `ld y` completes. Delaying `st x`, delays `st y` until `ld y` completes.

Second, delaying the write of x by withholding the response to its invalidation will delay the write on y even if this write is done by a third core, as long as x and y are updated in a transitive happens-before order dictated by program-order and synchronization-order. See the example in Table 3.

Third, if `st x` and `st y` are on different cores and independent, i.e., their happens-before order is established purely by chance and it is *not dictated by program-order or synchronization-order*, delaying `st x` has no effect on `st y` and does not prevent `ld y` from seeing the new value of y. However, since there is no program-order or synchronization-order to enforce between the stores, *the stores can be swapped in memory order*. In fact, delaying the invalidation response to `st x` will move `st x` *after* `st y` in memory order, yielding a legal TSO interleaving in the case where `ld y` reads the new value of y.

Fourth, a *miss-under-miss* scenario where the read requests are *reordered* in the network, is equivalent to the hit-under-miss scenario. If the younger load, `ld x`, reads the *old* value of x, the underlying protocol (with 3-hop read transactions with `Unblock` [26]) ensures that, in that case, `ld x` gets a cached copy of x (that will be invalidated) *before* `ld y` can see the new value of y. This makes it the same as the hit-under-miss scenario. The *hit-under-miss* (Figure 1) and *miss-under-miss* are two cases that can lead to the {new, old} case in Table 2. If both loads hit, or if the older load hits but the younger misses, there is no such danger.

3 LOCKDOWNS AND WRITERSBLOCK

In the previous section we show that, to preserve TSO, it is not necessary to squash a reordered younger load upon receiving an invalidation —it suffices *not to return* an acknowledgment until the time that the older load is performed. The mechanism to achieve this is the *lockdown*: When a load performs out-of-order with respect to any older load in the same core, it *locks down*, i.e., *it will not acknowledge invalidations* until all previous loads perform.

However, a lockdown by itself is not enough. In addition, we need to guarantee that:

(1) A store is blocked (not made globally visible) until *all* existing lockdowns for the store's cacheline address, on all cores, are lifted.
(2) No further writes for the address in question can take place in memory order before the blocked store is allowed to be performed.

This sets an upper bound for the delay, which is still less than typical *timeouts* to detect errors (e.g., dropped messages) which work at a much larger timescale; we assume that properly adjusted timeouts pose no problem for our approach.

	ordered w.r.t.l	unordered w.r.t.l
not performed	*SoS Load*	—
performed	*Completed*	*M-Speculative* (lockdown)

Table 4: Terminology for loads. "*Ordered w.r.t.l*" means ordered with respect to loads. "*SoS*" means Source of Speculation.

completed ←			→ unordered				
p	p	SoS (n-p)	n-p	n-p	M-speculative (p)	n-p	...
Head		LQ (Program order →)					Tail

Table 5: SoS load. "p": performed; "n-p": non-performed;

(3) Loads are never blocked, so that the lockdowns can be lifted to unblock the store.

We achieve these goals by introducing a new *transient* directory state that blocks a coherent write request from completing until the relevant lockdowns are lifted, yet at the same time, never blocks the loads' read requests from accessing the current value of the data. This new transient coherence state is called *WritersBlock* and we refer to the resulting coherence protocol as the WritersBlock protocol. Typically, transient directory states for writes block both new reads and new writes. In WritersBlock we *decouple* read blocking from write blocking and enforce only the latter.

WritersBlock coherence is an extension of a typical, invalidation-based, MESI directory protocol (e.g., as in GEMS [26]) to support cores that can set lockdowns, while maintaining compatibility for cores that use squash-and-re-execute to enforce TSO on speculative reordering.

3.1 Setting the Stage

In this paper we distinguish loads and stores from reads and writes. Loads and stores are instructions (that issue, execute and commit) while reads and writes are transactions of the coherency protocol. They differ in granularity (loads and stores operate on bytes, words, etc., while reads and writes operate on cachelines) but more importantly they differ on how we consider them to be performed.

3.1.1 Loads and Reads. Table 4 summarizes our terminology for loads. We say that a load is *performed* when it receives data; prior to that, the load is *non-performed*. We say that a load is **ordered with respect to loads in its core** when all prior (older) loads in program order are performed; otherwise the load is *unordered*. Since we only focus on load-load reordering, in this paper we shall use the term *ordered* to mean *ordered with respect to loads* and differentiate otherwise.

An ordered and performed load is *completed*. A load can be ordered without necessarily having been performed. This is a special and important case. There can be only one such load in the core. We designate it as the *Source of Speculation* or *SoS load*. It means that all loads before it are completed and all loads after it are unordered (see Table 5).

Finally, unordered loads that are performed are *M-speculative* loads [14]. A load goes into *lockdown* when it becomes M-speculative, i.e., *it is performed out-of-order with respect to older non-performed loads*. A load exits lockdown when it becomes *ordered*.

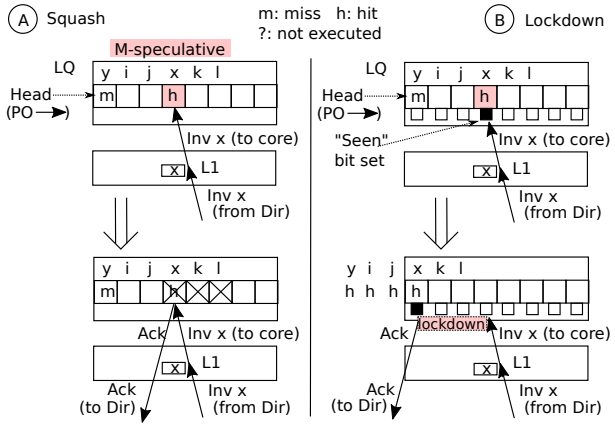

Figure 2: Lockdown concept. A: squash-and-re-execute core; B: lockdown core

A load can be speculative for other reasons. We define a load to be *control-speculative* (C-speculative) if it is on a speculative path (i.e., past unresolved branches); and *dependency-speculative* (D-speculative) if prior stores have unresolved addresses. If a load is squashed as C-speculative or D-speculative, its lockdown is *ended*.

A *read* is the coherent request of a load at *cacheline granularity* (often denoted as a `GetS` transaction). We say that a *read* is *performed* when it brings the data in the L1.

3.1.2 Stores and Writes. Stores enter the FIFO store queue (SQ) when they issue. They commit in program order when they have both the address and the data. The SQ keeps the correspondence of stores to the loads in the LQ. It allows stores to commit when all prior loads are ordered. We are not relaxing the load→store order as it only delays the commit of stores which has no significant performance impact in TSO.

Committed stores leave the SQ and enter the FIFO Store Buffer (SB) that enforces TSO's store→store. We say that a store is *performed* when it makes its value globally visible by writing it in the cache. A store writes the cache when it reaches the head of the SB *and* has write permission.

Request for write permission can occur as early as the store resolves its address. A coherent write request, at cacheline granularity (typically a `GetX` or `Upgrade`), prefetches the cacheline (if needed) with write permission.[3] We say that a write is *performed* when it gets the data with write permission in the cache. Writes are not bound by the program-order of the stores that initiate them and can be performed in any order.

3.2 Lockdowns

Figure 2 contrasts the operation of a squash-and-re-execute core, (Figure 2.A) with one having a lockdown mechanism (Figure 2.B). The example uses the `ld ra,y` and `ld rb,x` instructions from our previous examples but with additional loads interspersed in-between. We depict only the core's LQ which keeps loads in program order (PO).

LQ entries are tagged with the load address and show the status of the instruction: `h`: hit, `m`: miss. The speculatively performed load is shaded light-red. Loads exit the head of the LQ (FIFO) at the same time they commit and are removed from the reorder buffer (ROB).

As it is shown in Figure 2.A, an invalidation for address `x` results in the immediate squash of `ld x` and all following (younger) loads in the LQ. Cache line `x` is invalidated; further accesses to `x` miss and go to the directory.

Figure 2.B, depicts the lockdown approach. Each LQ entry is augmented with an extra "seen" bit (S). The S bit is set when some other core sees this load execute out of order, i.e., when an invalidation matches the target address of the load while the load is in *lockdown*. In contrast to the squash-and-re-execute case, loads are not squashed upon invalidation.

The S bit withholds the `Ack` for the invalidation until the time the load *exits* the lockdown mode, i.e., becomes ordered (all the loads that precede the load are performed) or is squashed (if it is C-speculative or D-speculative and we have a misspeculation). At this point if the S bit is set, we return the acknowledgement to the invalidation.

We can potentially have as many lockdowns as unordered loads in the LQ, irrespective of their target address. In fact, if we have multiple M-speculative loads on the same (cacheline) address, they can be all in lockdown. In this case, the S bit is set for the *youngest* load. If this load is squashed because it is C-speculative or D-speculative, the S bit is transferred to the next youngest load that survives the squash. Only when the youngest load becomes ordered (or is the last one for this cacheline address to be squashed) the invalidation `Ack` is returned.

3.3 WritersBlock: Block the Writes

With the WritersBlock protocol we aim to achieve three goals: First, we need to put the directory entry in a state that will hold the write in waiting until all the lockdowns are lifted. Second, we must block all new writes. Third, at the same time, we must allow new reads (loads) that reach the directory to see the current value of the data, i.e., the last value before the pending write.

Figure 3 contrasts the base protocol for writes with the modifications that we implement in the WritersBlock protocol (highlighted in red). Figure 3 shows the directory, the writer, and one sharer (only the LQ of the sharer is shown). There may be more sharers (not shown) or just the one shown in exclusive or modified state. In the lockdown case (Figure 3.B) the sharer sets a lockdown on the address.

Base protocol: Upon a write miss the writer sends a write request to the directory (Figure 3.A, step ①). The directory blocks for reads *and* writes to the target line until the write transaction completes. The directory sends invalidations (`Inv`) to the sharers (Figure 3.A, step ②). Invalidation acknowledgments (`Ack`) are returned to the writer (Figure 3.A, step ③). The writer gets the data either from the directory (it has an up-to-date copy) or from the single exclusive/modified copy. In this case, the exclusive/modified copy sends the invalidation acknowledgment and the data in the same message

[3] If write permission has been lost by the time the store reaches the head of the SB, the store requests write permission again and writes the cache before it relinquishes the permission.

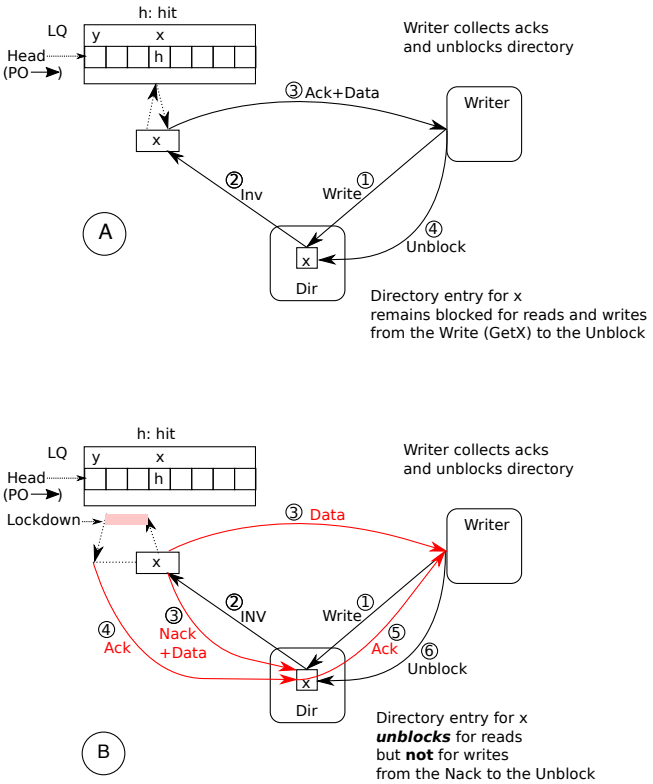

Figure 3: Write that hits a lockdown in a core

(Ack+Data). When the writer has the data and all the acknowledgments (all sharers have been invalidated) it unblocks the directory (Unblock).

WritersBlock protocol: *The common case of a write that does not hit a lockdown remains unchanged from the base protocol.* However, an invalidation that hits a lockdown sends a Nack to the directory (Figure 3.B, step ③).

It is this Nack that puts the directory entry into the *WritersBlock* state which blocks all writes but allows reads to proceed.

An important detail here is the way we guarantee that the directory supplies the correct data. The shared level (e.g., LLC) may have stale data when there is a single exclusive or modified copy that is invalidated. In this case, the data are simultaneously sent to the shared level along with the Nack (Nack+Data) and to the new writer (Data), as depicted in Figure 3.B, step ③. Thus, the readers are provided with a place to access to the data, as the exclusive copy is no longer accessible via the directory —it has been invalidated— and the new writer is not visible yet.

When a lockdown is lifted, an Ack must be returned to the writer. Lockdowns do not retain the identity of the writer as there is only one S bit to indicate that an invalidation has occurred but not who is invalidating. The Ack is redirected to the writer via the directory entry where the writer's identity is known (Figure 3.B, steps ④,⑤). When the writer has the data and all of the acknowledgments it unblocks the directory.

To summarize, the major changes over the base protocol for writes are:

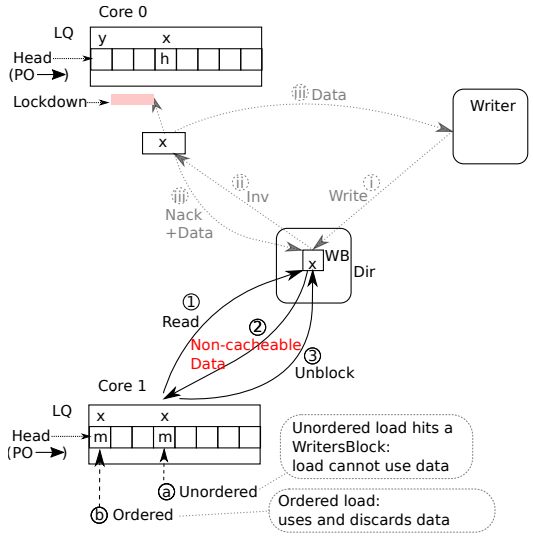

Figure 4: Read that hits a "WritersBlock" directory entry

- A lockdown returns Nack to the invalidation, putting the directory entry in the WritersBlock state that allows reads (of the latest "old" data) but blocks all writes. The current writer awaits the Ack to complete its transaction.
- The Ack of the invalidation takes more time and redirects through the directory to the writer.
- Data are sent to both to the directory and to the writer if the invalidated cache line was in exclusive state.

3.4 WritersBlock: Admit the Reads

The common case of a read to a directory entry not in WritersBlock remains unchanged from the base protocol. Reads are 3-hop transactions (e.g., GetS request, Data, Unblock), either finding the data in the shared level cache, or redirecting via the directory to the cache that has the exclusive copy. Here, we discuss what happens when a read finds the directory in WritersBlock.

Consider a load that misses in the cache, sends a read to the directory, and finds the directory entry in WritersBlock. Such a read is allowed to access the latest version of the data before the write, but cannot see the new value of the write. The read catches the write midway, but the outcome is identical as if the read happened *before* the write. Taking this view, the reader core should see no difference than if it was one of the initial sharers at the time of the write, had a copy of the current version of the data, *and* was invalidated by the write. WritersBlock implements this behavior. There are two ways to do this:

Option 1: Let the reader cache a copy of the data and send a new invalidation. Consider what would happen if we opted to return *cacheable* copies to the reads: The directory has already sent all its invalidations for the initial sharers and is waiting for an Unblock from the writer. If new read requests obtained cacheable copies, they would have to be invalidated *before* the writer is allowed to perform and Unblock the directory. The directory would have to go into another round of invalidations for the newcomers.

The danger is that this can continue in perpetuity and *livelock*. Spin loops waiting for a write to set a value are the prime example: new read requests arrive at a rate that forces the directory to indefinitely delay the write by having to constantly invalidate new sharers. Obviously, we cannot consider this option further.

Option 2: Serve an *uncacheable* version of the data without registering the reader in the sharing list; i.e., an *uncacheable tear-off* copy of the data [23]. In this way, there are no new sharers to invalidate.

Figure 4 depicts the uncacheable tear-off protocol for reads. Core 1 issues a read request for a load that misses (Figure 4, step ①). The directory, in WritersBlock, replies with an *uncacheable* copy of the data that can be used at most once (Figure 4, step ②). Since the copy is not cached, the directory does not track it.

Consider, however, what happens when the load instruction that causes the read is the unordered load ⓐ in Figure 4. If load ⓐ uses the tear-off copy to perform and become M-speculative, it should prevent the writer from performing. It is not correct for the writer to be performed as this allows load ⓑ, the SoS load, to see a new value. As we explain in Section 2, this violates TSO. The problem is that there is no mechanism for load ⓐ to maintain the WritersBlocks until load ⓑ is performed, as there is no invalidation Ack to delay.

For this reason, load ⓐ is *not performed* and must repeat the request anew. We optimize this case by only repeating the request when the load becomes ordered, i.e., when it becomes the SoS load. It is always correct for the ordered SoS load (e.g., load ⓑ) to be performed immediately using the uncacheable tear-off copy.

As a further optimization, a core refrains from issuing new (unordered) loads for any address for which there is already a lockdown *and* an invalidation has been received. We know, in this case, that such new loads in this core will receive uncacheable tear-off copies that they cannot use.

To summarize, we choose to implement Option 2 for reading from a WritersBlock state as this is a livelock-free solution. Our solution has an important implication:

> There are *no new lockdowns after invalidation*: A write can only be blocked by a *fixed* number of loads that are in lockdown at the time of invalidation. New loads in the invalidated cores and new loads in newcomer cores are not allowed to block anew an already blocked write.

3.5 Ensuring Safe Passage for SoS Loads

WritersBlock prevents stores from being performed by blocking their write transaction until the lockdowns that caused the WritersBlock are lifted. As we have explained, lockdowns are lifted when the M-speculative loads become ordered. The key to understanding how deadlocks can be avoided is the following observation: *The ability of an M-speculative load to become ordered hinges solely on the ability of the SoS load to be performed—whichever the SoS load might be at any point in time.* If the SoS load is blocked because of a WritersBlock, a deadlock ensues. The condition that we need to guarantee to avoid deadlocks it the following:

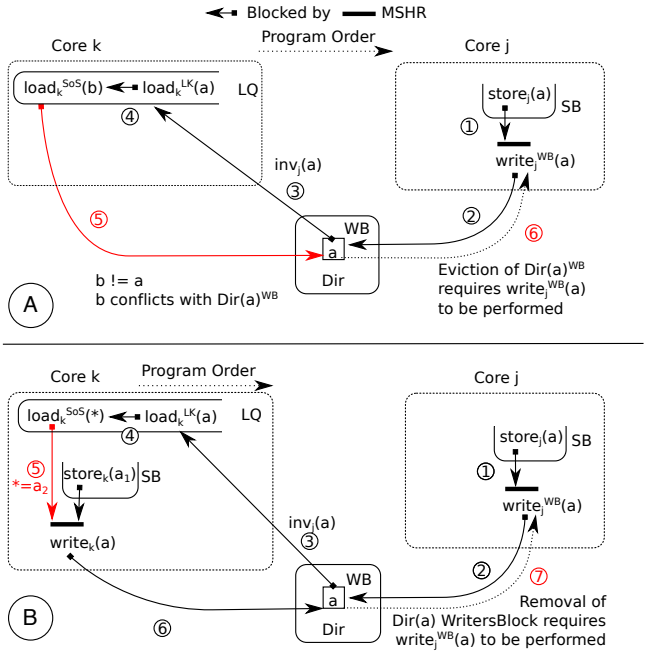

Figure 5: (A) Directory deadlock, (B) MSHR deadlock.

> *A SoS load cannot be blocked anywhere in the memory system.*

WritersBlock changes the behavior of the directory and the behavior of the write transactions. This creates only two situations where a SoS load can be blocked:

- attempting to bypass a WritersBlock entry at the directory; and
- attempting to bypass writes that are blocked in the MSHRs of their core.

We explain with examples how these two situations can possibly give rise to *deadlocks* and we describe our measures to avoid them. To discuss deadlocks, we use a *blocked-by* dependence: operation x is blocked-by y, if y prevents x from being performed. Figure 5 shows the two deadlock scenarios: one at the directory and one at the MSHRs.

3.5.1 Safe Passage through the Directory. The purpose of the WritersBlock state in a directory entry is to block writes but allow reads to proceed. This is guaranteed for reads that access the same address as the directory entry.

However, a WritersBlock on address a can inadvertently block a read on a *different* address b. This could happen when a read on b needs to evict directory entry a (which is in WritersBlock).

Figure 5.A shows this case: a reader core (core k) and a writer core (core j) deadlock because of the eviction of the WritersBlock directory entry. Let us follow the example[4] to see why this can happen:

[4]In the figure, program order is from left (older) to right (younger). Addresses for stores and loads refer to the size of the register loaded or saved, but addresses for write misses (e.g., GetX, Upgrade) and read misses (e.g., GetS) refer to the cache line size that contains the item loaded or saved. In other words, false sharing is taken into account. Blocked-by dependencies are depicted by an arrow with a dot on its tail.

A store on address a in core j (represented as $\texttt{store}_j\texttt{(a)}$) requests write permission with a write request: $\texttt{write}_j^{\texttt{WB}}\texttt{(a)}$ which is blocked by the lockdown of a load on address a in core k ($\texttt{load}_k^{\texttt{LK}}\texttt{(a)}$) via the WritersBlock state $\texttt{Dir}^{\texttt{WB}}\texttt{(a)}$, (steps ①,②,③).

The question is whether the SoS load in core k is guaranteed to be performed. If the SoS load's address resolves to b, different than a, and b conflicts with a in the directory, we have a cycle (and deadlock). The directory entry cannot be evicted until the WritersBlock is lifted. This in turn depends on $\texttt{write}_j^{\texttt{WB}}\texttt{(a)}$ being performed and $\texttt{write}_j^{\texttt{WB}}\texttt{(a)}$ cannot be performed unless the directory entry is evicted to allow the SoS load, $\texttt{load}_k^{\texttt{SoS}}\texttt{(b)}$, to be performed.

A variation of the above scenario is when there is no writer. The eviction of Dir(a) causes the directory entry to enter WritersBlock through the *eviction*-invalidation it sends to core k that finds $\texttt{load}_k^{\texttt{LK}}\texttt{(a)}$ in lockdown. In this case, the WritersBlock must remain until the lockdown is lifted, otherwise a future writer might miss the lockdown as there will be no way to reach it after the eviction of the directory entry.

Solution: *Instead of trying to evict a WritersBlock entry, the load simply obtains an uncacheable tear-off copy of the data and performs without needing a directory entry.*

Increased use of uncacheable tear-off copies (especially when they must be brought from memory), could manifest as a performance problem as the effectiveness of caching is reduced. However, the standard practice for evictions renders it a non-issue: Eviction is performed on the side, in an eviction buffer (e.g., MSHRs). This allows a load to immediately claim its directory entry and obtain a *cacheable* copy of the data. The WritersBlock entry under eviction is put in the eviction buffer until it is safe to discard, *after* the blocked write completes.

It is only in the rare case when the eviction buffer is full, and the read cannot allocate a directory entry, that the read turns into an uncacheable transaction. Deadlock is thus avoided while uncacheable reads are practically eliminated.

3.5.2 Safe Passage through the core's MHSRs. The second situation where a SoS load could be potentially blocked is when it tries to bypass blocked writes in its core's MSHRs. TSO allows loads to bypass stores in the store buffer, but there is a problem with their writes. The problem lies with a common optimization that coherence protocols do for simplification. If a load's *cacheline* address matches a store's *cacheline* address and the store has an outstanding write request in the memory system, the load piggybacks on the same MSHR and awaits the resolution of the outstanding write request.[5] The write request, however, can be blocked in WritersBlock.

Figure 5.B shows how a deadlock can arise in this case. As in the previous example (Figure 5.A), $\texttt{write}_j^{\texttt{WB}}\texttt{(a)}$ is blocked by $\texttt{load}_k^{\texttt{LK}}\texttt{(a)}$ in core k (steps ①,②,③). Another write from core k, $\texttt{write}_k\texttt{(a)}$, is blocked in its MSHR by $\texttt{Dir}^{\texttt{WB}}\texttt{(a)}$ in WritersBlock.

Assume now that the SoS load in core k, $\texttt{load}_k^{\texttt{SoS}}\texttt{(*)}$, resolves its unknown address "*" to a_2 which is in the same cacheline a as a_1 and piggybacks on the same MSHR used by $\texttt{write}_k\texttt{(a)}$ for this cacheline (step ⑤). The SoS load will not be performed, as it is blocked by $\texttt{write}_k\texttt{(a)}$, which in turn is blocked by $\texttt{write}_j^{\texttt{WB}}\texttt{(a)}$ (in WritersBlock at $\texttt{Dir}^{\texttt{WB}}\texttt{(a)}$), blocked by the lockdown of $\texttt{load}_k^{\texttt{LK}}\texttt{(a)}$, which is prevented from lifting until $\texttt{load}_k^{\texttt{SoS}}(a_2)$ *itself* is performed.

Solution: *A SoS load launches a read on a new MSHR to bypass a potentially blocked write.*

It is only when a load becomes the SoS load, we need to launch a read on a new MSHR. As a further optimization, we return a hint to any write that enters or encounters a WritersBlock (the hint message is not shown in Figure 3), so we *know* when a write is blocked. A SoS load initially piggybacks on the MHSR of a write and waits. If and when it is determined that the write has blocked, a new read is launched on a new MSHR. This new read receives an uncacheable tear-off copy as described in Section 3.4.

As our approach may require a new MSHR to be allocated for the SoS load, we assume *resource partitioning* at all levels of the hierarchy: There is at least one MHSR always reserved for SoS loads; stores or evictions cannot hog all MHSRs.

3.5.3 Summary. Deadlocks could arise because of resource conflicts or resource exhaustion: i.e., when the read of a SoS load conflicts with a blocked write on the same MHSR, or when it conflicts with a directory entry in WritersBlock. We have a simple strategy to avoid these deadlocks:

> *SoS loads bypass the resources (MSHRs and directory entries) on which they can block; their reads become uncacheable.*

As a result, SoS loads cannot be blocked by writes, and consequently by stores, directly or indirectly, anywhere in the memory system and are guaranteed to be performed. This means that lockdown loads are guaranteed to become ordered.

3.6 Ensuring that stores are performed

A store can only be blocked via its write request by the lockdown loads of the cores it invalidates. Their number is fixed, since we allow no new lockdowns for an address in WritersBlock. Since lockdown loads are guaranteed to become ordered, writes are also guaranteed to be performed.

Stores are guaranteed to be performed even though they must be performed in program order (i.e., when they reach the head of the FIFO store buffer), but can send their write request in any order (e.g., to prefetch write-permission). The reason is that it is the *completion of the write transaction* that lifts the WritersBlock —not the *store being performed*. These two events are decoupled: A store may be performed only when it reaches the head of the FIFO store buffer (and still has write permission), but its write may be sent earlier and be performed without any ordering restriction with respect to other writes in the same core. This ensures that stores from different store buffers do not deadlock when sending write prefetch requests.

3.7 Atomics

Atomic read-modify-write (RMW) instructions represent a special case for all consistency models and out-of-order architectures [35] —it is not different in our case. An atomic RMW instruction is an atomic load-store pair. In TSO, the load of an atomic instruction is not allowed to bypass stores in the store buffer. This would violate

[5] Of course, as per TSO, if the load's address matches exactly the store's address, the load takes on the store's value available in the store buffer.

either the store→store order or the atomicity of the instruction [35]. Thus, the store buffer needs to be drained for the atomic instruction to execute. In WritersBlock coherence, however, in order for the store buffer to drain, it may be necessary for the load of the atomic data to bypass blocked stores in the store buffer.

Furthermore, even if the store buffer drains without a problem, the load of an atomic instruction behaves as a store. In fact, in many implementations the load issues a write transaction to obtain write permission. This means that the load itself can block in WritersBlock.

Thus, the load of an atomic RMW violates the basic premise of our approach, that SoS loads cannot be blocked. In other words, the load of an atomic RMW can never be a source-of-speculation (SoS) load. This means that no load following an atomic instruction in the ROB can go into lockdown mode. If we allowed this, deadlock could easily ensue.

For this reason, we default to the behavior of the baseline core architecture: If the underlying core supports squash-and-re-execute, loads following an atomic instruction can issue before the atomic instruction executes, but may get squashed. If the underlying core does not support squash-and-re-execute, loads cannot issue until the atomic instruction executes. In this case, however, prefetches can be issued instead and loads can use the prefetched values only after the atomic instruction is performed.

3.8 Cache Evictions

With respect to speculative reordering in TSO, we must distinguish between evictions that do not remove the evicted line from the directory's sharing list (e.g., as in OpenPiton [4]), and evictions that remove the evicted line from the sharing list. We will call the former *silent evictions* and the latter *non-silent evictions*.[6] Various protocols implement one of the two eviction methods or even both, choosing between them based on the evicted line's state.

Non-silent evictions in the baseline protocol must cause a squash of M-speculative loads and all instructions that follow. The reason is that if a line is evicted, it will not be notified if it is written: the directory will not send an invalidation to a non-sharer. This can lead to a TSO violation in the example of Table 1 and Figure 1. Conservatively, a non-silent eviction squashes M-speculative loads in the off-chance that a write would occur in the reordering window.

Silent evictions, on the other hand, do not query the LQ, minimizing squashes. The downside, in this case, is that we may have invalidations that do not find a cache line in the L1 —not possible with non-silent evictions— which also must query the LQ.

Depending on the eviction type in the baseline protocol, in our approach:

- Silent evictions in the baseline remain silent.
- Non-silent evictions in the baseline that do not cause a squash remain non-silent.
- Non-silent evictions in the baseline that cause a squash, i.e., under a lockdown in our approach, become *silent* instead of squashing. This guarantees that a write during the reordering window will observe the lockdown and block accordingly. In the remote case of an eviction under a lockdown, we trade off a squash with the possibility of an extra invalidation that must reach the LQ even in the absence of a hit in the cache. This tradeoff is to our advantage.

We chose our baseline protocol with silent evictions for shared lines, resulting in 9.6% lower traffic (25% lower in some benchmarks) and with similar performance as a baseline with non-silent evictions [17].

4 USE CASE: OUT-OF-ORDER COMMIT

Our motivation for non-speculative load-load reordering in TSO is the potential for irrevocable binding of loads. This opens up a slew of possibilities, from non-speculative execution to out-of-order commit, past a non-performed load. In this paper, we use out-of-order commit to demonstrate the performance benefits of our approach.

Speculative vs. safe out-of-order commit: In general, there are two classes out-of-order commit proposals: those that use extra, *uncore* (outside the core) checkpoint-and-rollback mechanisms to allow *speculative retirement* [7, 11–13, 22, 27, 29]; and those that only commit when it is *safe to do so* [1, 5, 15, 21, 25, 37, 38].

Checkpoint-and-rollback approaches extend the speculative mechanisms of a processor beyond the dynamic instruction window. Allocating uncore checkpointing and rollback resources defeats the purpose of using out-of-order commit for energy-efficiency and drastically complicates the designs. Invariably, such work grapples with reducing the excessive cost and complexity of speculation [7, 13].

In contrast, a more conservative approach of *safe* out-of-order commit has the potential for higher efficiency as it introduces no additional checkpointing cost. The safe approach was articulated in the work of Bell and Lipasti [5] in the form of six limiting conditions. The necessary conditions to allow an instruction to safely commit out-of-order are:

(1) The instruction is completed. Obviously, instructions can commit only after their completion.
(2) Registers: Register write-after-read (WAR) hazards are resolved (i.e., a write to a particular register cannot be permitted to commit before all prior reads of that architectural register have been completed).
(3) Branches: Previous branches are successfully predicted. This condition simply says that we can commit only while on the correct path of execution.
(4) Stores: We cannot commit speculative loads (or their dependent instructions) if an earlier store has an unresolved address (unknown memory dependence).
(5) Exceptions: No prior instruction in program order is going to raise an exception.
(6) Consistency: We cannot commit loads and stores unless the global order is not perturbed in a way that violates the consistency model: i.e., the proper memory order for all previous memory operations is already established.[7]

Often, the first five conditions resolve fast (on the order of instruction latencies). The branch and the unresolved-store-address conditions typically resolve in a few cycles unless the corresponding instructions depend on a miss. The sixth condition, however, typically

[6] Albeit this naming convention may not accurately reflect the communication with the directory as this depends on the state (e.g., clean, dirty) of the data.

[7] This condition covers previous loads with unresolved addresses that were lumped in condition 4 by Bell and Lipasti [5].

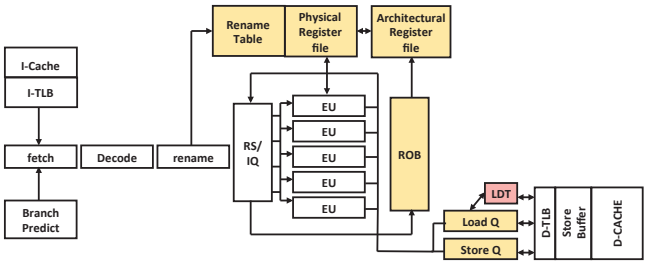

Figure 6: Block diagram of Out-of-order-commit microarchitecture. Shaded structures participate in out-of-order-commit.

requires significant time to resolve, as loads are reordered due to misses and consistency enforcement requires that we wait for these misses to be resolved.

Example: In an out-of-order commit core, for the same TSO example given in Section 1, we have the opportunity to commit the younger load and its dependent instructions, provided of course that they satisfy the rest of the conditions listed above (for example, the instructions are not on an incorrect path). However, *in an unordered network and with directory coherence*, until it is determined that the reordering has not been caught by anyone, they cannot commit: If instructions irrevocably commit out-of-order while transgressing the ordering rules of a consistency model (in this case TSO), there is no checkpoint-and-rollback to repair the damage if caught. Lockdowns and WritersBlock coherence address exactly this problem.

4.1 Out-of-order-commit microarchitecture

The out-of-order-commit microarchitecture model of this paper follows the Bell-Lipasti design [5]. As a base case we assume out-of-order commit only when it is safe to do so according to the six Bell-Lipasti conditions. A detailed analysis of the Bell-Lipasti conditions appears in [3].

Bell and Lipasti describe and study three techniques for the implementation of an out-of-order-commit microarchitecture: a collapsible ROB design that removes gaps left from committed instructions; a design that fills gaps with new instructions and keeps program order with an indirection; and a design that leaves gaps empty until they reach the head of the ROB where they are discarded. Performance analysis shows that closing or filling the gaps is necessary to obtain benefits [5]. From the first two options they settle on a collapsible ROB design as it naturally keeps program order in its entries as opposed to the alternative design that significantly complicates the commit logic (testing for the six conditions).

Figure 6 shows the block diagram of the core we model in this paper. Shaded structures are involved in out-of-order commit. In our base model, the ROB and the Load Queue (LQ) are collapsible, and the Store Queue (SQ) and the Store Buffer are FIFO. We add a small table ("LDT") next to the collapsible LQ (see Subsection 4.2).

4.2 Lockdowns for a Collapsible LQ

The benefits of out-of-order-commit come from committing loads and dependent instructions out-of-order allowing a collapsible ROB to bring in more instructions without stalling. However, a non-collapsible LQ may become a bottleneck as pressure increases from the collapsible ROB. Increasing LQ size might not be the best option (it is an expensive CAM), hence the *preference* is for a collapsible LQ. In this paper we consider a collapsible LQ, in which committed loads are removed from any position.[8]

At any time, there is only a small number of M-speculative loads that can commit out-of-order. The reason is that after committing a few loads out-of-order, it becomes increasingly likely that one of the Bell-Lipasti conditions will kick in to prevent further out-of-order commit [5]. If an M-speculative load commits out-of-order, it exports its lockdown to a small structure at the L1, called the Lockdown Table (LDT). Each committed load corresponds to an entry in the LDT. A small number of lockdowns (e.g., 32) is kept in the LDT and in the rare case we reach this limit, we stop committing M-speculative loads out-of-order. As in the case of a non-collapsible LQ, we allow multiple lockdowns in the LDT for the same cacheline address and we return an `Ack` (if there was an invalidation) only when the *last* lockdown in the LDT for this address is lifted. Similarly, the LDT allows multiple lockdowns for the *same cache line address* (one per load). On invalidation, the S bit is set for *all* LDT entries of the same address, but the `Ack` is sent only when the *last* lockdown in the LDT for this address is released. This corresponds to the same condition mentioned previously: Only when the youngest M-speculative load for a given address becomes ordered is the invalidation `Ack` returned.

Incoming invalidations search the LDT associatively using the cacheline address and set the "seen" S bit of the matching LDT entry. The key to correct operation is to release each lockdown and return the invalidation `Ack` (if the S bit is set) when the corresponding M-speculative load would have become ordered. To achieve this, having removed the committed load from the LQ, we assign the responsibility of releasing its lockdown to its immediate *older non-performed* load, (i.e., the first non-performed load towards the SoS load). If that older load also commits while being M-speculative, it passes all the lockdowns for which it is responsible (including its own) to the next non-performed load, and so on, until we reach the SoS load. When the SoS load is performed it lifts all the lockdowns it has been assigned. The lockdowns are efficiently encoded in the LQ entries as a bitmap index to the LDT entries.

The LDT endows a collapsible LQ with the lockdown functionality of a non-collapsible LQ.

LDT Example: Figure 7 shows the operation of the LDT in relation to a collapsible LQ. The example is the same as in the in-order commit case, shown in Figure 2. Each LDT entry contains the address of a lockdown, and the corresponding "seen" bit, S. When a load commits out-of-order it is removed from the LQ, for example, `ld x` in step ①. At this point the lockdown is transferred to the LDT by allocating a new entry, e.g, LDT[1].

The responsibility to lift the lockdown of `ld x` is passed to the first available *older* load in the LQ, `ld j`, by assigning it the index of LDT[1].[9] More than one index can pile up on a load. For example, in step ②, `ld k` commits, sets its lockdown in LDT[3], and passes the lockdown index to the LQ entry of `ld j`. Any load that is removed from the LQ transfers its set of indices (the bitmap that also encodes

[8] In the evaluation, we use the same size LQ (number of entries) in in-order commit and OoO-commit; the *effective size* in the collapsible LQ is larger as we free up committed loads.

[9] Although indices are encoded in a bitmap, in the figure we show them as individual entities for clarity.

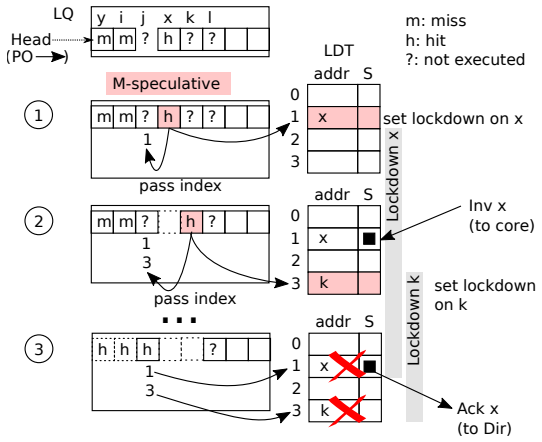

Figure 7: LDT/Collapsible LQ operation

its own index) to the first available LQ entry on its left. Between steps ② and ③, `ld j` "guards" two more lockdowns (1 and 3). At the moment it becomes *performed and ordered* in step ③, it lifts its lockdown if it had set one[10] and releases all the lockdowns in its set of indices. When lockdowns are lifted, `Acks` are sent for the invalidations that may have arrived in the interim. This is handled by the "seen" bit in each lockdown entry in LDT. `Ld x` which committed out-of-order is "seen" by an invalidation that matched the lockdown LDT[1] in step ②, and set the "seen" bit. When the lockdown is lifted in step ③, the invalidation is acknowledged.

5 EVALUATION

Our simulation infrastructure is based on the GEMS simulator [26], which offers a detailed timing model of the memory hierarchy, connected to a x86-like in-house out-of-order processor model that provides TSO and is driven by a Sniper [9] front-end. The interconnect is modeled with GARNET [2]. We run the applications from the SPLASH-3 [32] and PARSEC 3.0 [6] benchmark suites, with *simsmall* inputs, and present results for their parallel region.

We simulate a multicore processor consisting of 16 out-of-order cores. The WritersBlock coherence protocol extends the functionality of the directory-based protocol (MESI states) provided by GEMS, to support blocking of writes and delivering uncacheable read copies. The processor model has been extended to implement out-of-order commit (*OoOCommit*) with a commit depth equal to the size of the ROB. OoOCommit in a directory protocol respects the sixth Bell-Lipasti condition (consistency enforcement) and cannot commit a reordered load. OoOCommit enhanced with WritersBlock coherence relaxes the consistency enforcement condition, thus allowing reordered loads to commit immediately provided they satisfy the other five conditions.[11]

The purpose of this evaluation is to demonstrate that WritersBlock coherence introduces negligible overheads by seldomly delaying stores and utilizing uncacheable tear-off copies of the data. Since the performance of WritersBlock may be sensitive to the depth of the load queue, we compare a range of cores from efficient Silvermont-class (SLM-class) to higher-performing Nehalem-class (NHM-class) and Haswell-class (HSW-class). Details of the simulated architectures are displayed in Table 6.

[10] `Ld j` may become M-speculative when it performs, but since it was not taken out of the LQ it holds its own lockdown without allocating an external one in the LDT, exactly as in the non-collapsible LQ.

[11] In our experiments the exception condition is inactive.

Table 6: System configuration

Processor: SLM-class / NHM-class / HSW-class	
Issue and commit width	4
Instruction queue (IQ)	16 / 32 / 60 entries
Reorder buffer (ROB) size	32 / 128 / 192 entries
Load queue (LQ)	10 / 48 / 72 entries
Store queue (SQ), Store buffer (SB)	16 / 36 / 42 entries
Lockdown table (LDT)	32 entries
Memory	
Private L1 cache	32KB, 8-way, 4 hit cycles
Private L2 cache	128KB, 8-way, 12 hit cycles
Shared L3 cache	1MB per bank, 8-way, 35 hit cycles
Memory access time	160 cycles
Network	
Topology / routing	2D mesh / Deterministic X-Y
Data / Control msg size	5 / 1 flits
Switch-to-switch time	6 cycles

5.1 Coherence Protocol Implications

WritersBlock guarantees that M-speculative loads are never squashed. Our goal is to achieve this with minimum overhead.

Delayed writes. In WritersBlock, when a write-request invalidation finds a M-speculative load, the write request is delayed until the load becomes ordered. If the ratio of delayed write requests is high, store latency can increase, adding pressure on the store buffer and causing slow-downs. Figure 8 (top graph) shows the number of write requests that are blocked per thousand of stores (kilo-stores) when varying the aggressiveness of the core. As expected, the NHM-class and HSW-class cores see more write-request blocked because of their larger LQ, but their ratio is still very low (0.4 writes blocked per thousand store operations). The worse case is *streamcluster*, where less than 1% of stores issue writes that block at the LLC.

Read misses. When write requests are in WritersBlock state, read misses get uncacheable data, that can be used by the load in case it is ordered. Responding with a vast amount of uncacheable data would increase the number of cache misses. Figure 8 (bottom graph) plots the number of uncacheable data responses per thousands of loads (kilo-loads). Again, a larger LQ implies more uncacheable reads, but its number is very low (less than 0.9 uncacheable reads per thousand load operations in the worst case *freqmine* and around 0.1 per kilo-loads on average).

Time and traffic overhead. Write delays are not frequent in WritersBlock. Even in aggressive cores with a large LQ, there is no perceptible difference in store latency. The reason is that aggressive cores have also a large store buffer able to tolerate extra latency while prefetching the write permission for the block. Therefore, the execution time does not vary due to delayed write requests. Differences in execution time are therefore minimal –Figure 10 (top graph). On the other hand, the overall impact on coherence traffic by the WritersBlock protocol is also minimal, since the number of extra cache misses introduced by WritersBlock is negligible –Figure 9 (bottom graph).

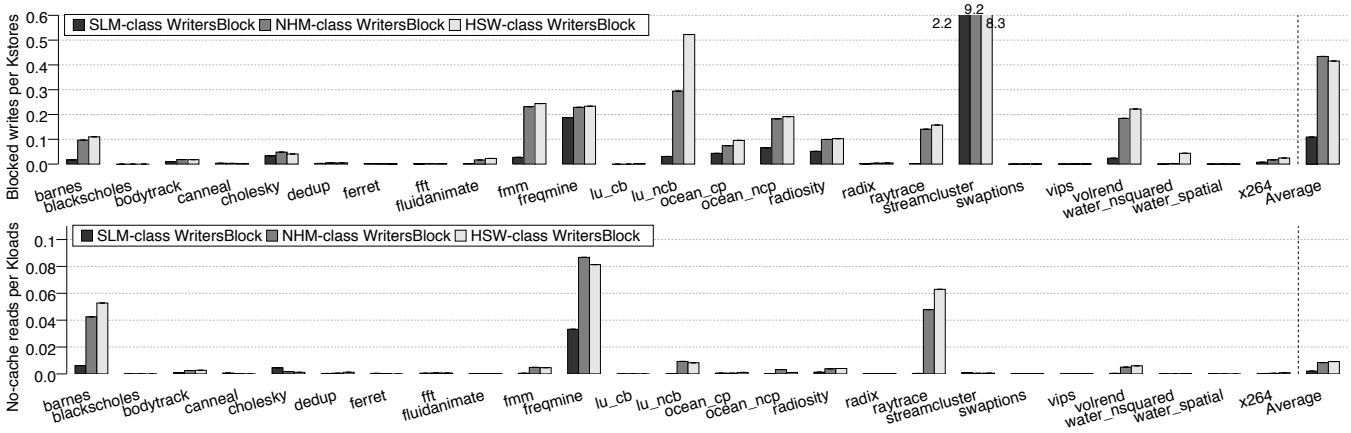

Figure 8: WritersBlock per kilo-stores and uncacheable reads per kilo-loads

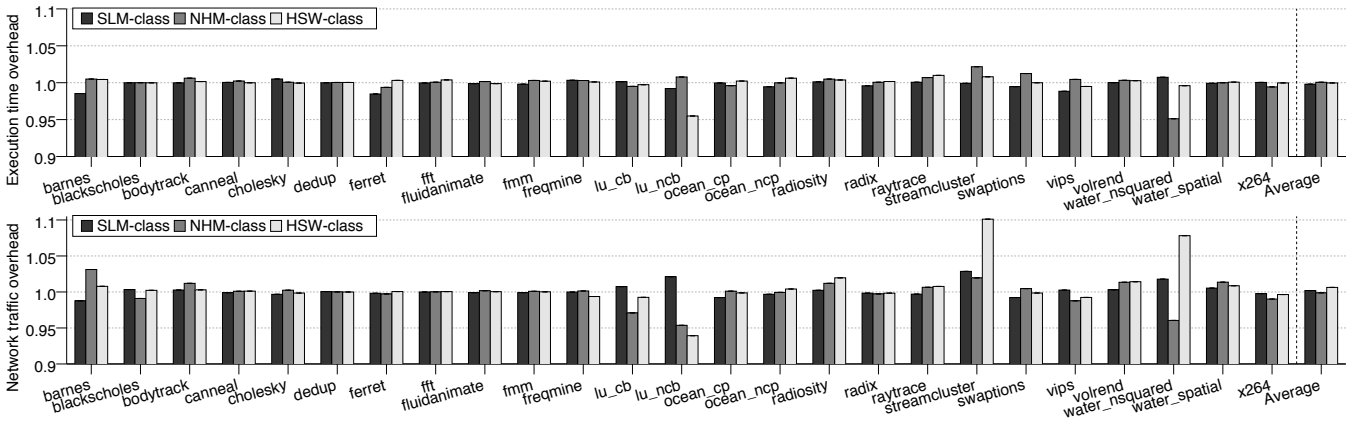

Figure 9: WritersBlock overheads: execution time and network traffic

5.2 Out-of-Order Commit Implications

While for in-order commit WritersBlock shows neither benefit nor penalty, the advantages available to a core that implements safe (non-speculative) out-of-order commit are high. This section shows these advantages for the SLM-class processor.

Processor stalls. Figure 10 (top graph) depicts the average percentage of cycles per core when cores stall (i.e., cannot commit a single instruction) and the reason why (SQ, LQ, or ROB full). One significant trend in OoO commit is the reduction in stalls due to a full ROB, because of the ability of committing OoO. However, in traditional OoO, the LQ becomes the bottleneck. Thanks to the commit of loads both earlier and out-of-order, this bottleneck is reduced in WritersBlock. On average, WritersBlock with OoO commit significantly reduces processor stall cycles.

Execution time: Due to the reduction of processor stalls, execution time is greatly improved by the combination of WritersBlock and out-of-order commit (14% on average, as seen in Figure 10 (bottom graph). In fact, the maximum benefit over in-order commit, seen by *bodytrack*, is 41.9%. Other applications such as *fft*, *lu_ncb*, and *ocean_ncp* also experience similar improvements. On average, WritersBlock with OoO commit shows 15.4% improvements over Directory-based coherence coupled with out-of-order commit. Over a directory protocol with OoO commit, that is, without our solution for safe OoO commit of loads, our simulations show improvements by 10.2%, on average, and up to 28.3%. Given these results we can conclude that the advantages of out-of-order commit can be enhanced with WritersBlock coherence to guarantee that speculative loads will not require re-execution.

6 RELATED WORK

Out-of-order commit: Committing instructions in program order may lead to significant performance degradation if a long latency operation blocks the ROB head. Several proposals have been published to deal with this problem in either a speculative, or non-speculative manner.

Speculative Retirement: Most works dealing with early retirement do so *speculatively* [11–13, 22, 27, 29]. Speculative retirement requires processor checkpoints to rollback the processor to a valid state. These works shift the speculation burden away from the main, expensive-to-scale structures for out-of-order execution, and instead focus on minimizing the overall cost of checkpointing.

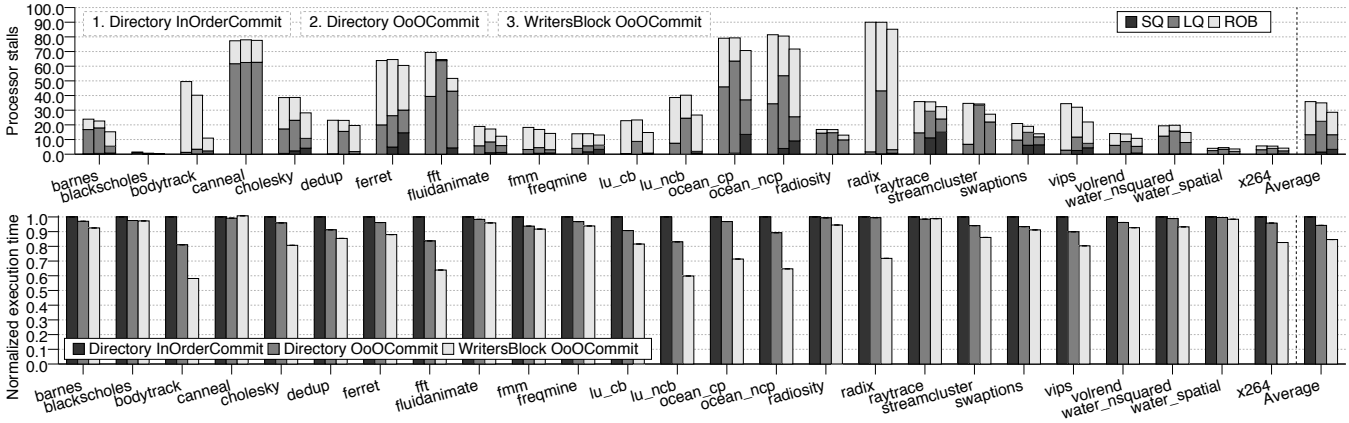

Figure 10: Percentage of processor stalls and normalized execution time with out-of-order commit

Non-speculative Retirement: Non-speculative early release of hardware structures before commit requires knowledge that no older instruction (that has come before the instruction to be released) can cause the program to abort, raise an exception, or require exposure of the architected state at that time. Non-speculative solutions [1, 5, 15, 21, 25] have the potential to be the most energy efficient, a necessity in an era of the end of Dennard Scaling and power-limited platforms.

Improving core performance through coherence: In Atomic Coherence [39], the authors present a globally-atomic method for coherence without the need for speculation. While this work is close to our work, it targets centralized, bus-based systems (based on optical interconnects), and does not directly apply to directory-based schemes to allow for fast and scalable resolution for ownership and load→load ordering.

Many recent works [7, 8, 10, 14, 18, 28, 29, 31] use speculation between the core and coherence models to improve performance. One major drawback of these works is the requirement for the core to squash and re-execute instructions in the case that an illegal memory reordering does occur. Our work, instead, uses the coherence protocol to *prevent* a reordering from being seen.

TSO-Driven Cache Coherence: Several recent works combine the coherence protocol with the consistency model: TSO-CC [16], Tardis 2.0 [40], and Racer [30]. We are inspired by these works which, however, still operate on the premise of squash-and-re-execute on consistency violations. We go a step further in this direction.

Non-speculative reordering: *Conflict Ordering* [24] characterizes the situation under which a younger memory operation can commit non-speculatively before an older memory operation. The authors use the idea to reorder store→load and store→store non-speculatively in SC. Subsequently, Gobe and Lipasti extend the conflict ordering concept to also perform load-load reordering in SC but they rely on taking mutexes for every address involved in the reordering [19]. Our work makes the observation that in TSO we can simply delay a conflicting store. This obviates the need for complete tracking of all memory operations of a core in an Augmented Write Buffer as is done in [24] or the use of global address mutexes as in [19].

Further, our approach has a fundamental advantage for load-load reordering not found in any earlier work: we can reorder over unresolved load addresses, whereas previous approaches require address computation.

7 CONCLUSION

In this paper, we present a novel cache coherence solution that can hide speculatively reordered loads in TSO so that a memory reordering in one core is not seen by other cores. The value loaded will always respect the load→load program order. This means that loads that execute out-of-order do not have to be squashed and re-executed due to consistency enforcement. As a consequence, loads can be committed out-of-order safely with respect to consistency enforcement. We demonstrate our approach for out-of-order commit and show significant improvements over the base case without our solution. Prior out-of-order commit proposals support TSO with expensive checkpoint-and-rollback mechanisms, or rely on globally ordered networks (e.g., snooping busses), or penalize performance by delaying the commit of reordered loads. As far as we know, this is the first solution for irrevocable binding of speculative loads under TSO, and the first solution allowing their out-of-order commit on architectures with general unordered interconnection networks and directory coherence.

ACKNOWLEDGEMENTS

We wish to thank Daniel Sorin for his invaluable guidance in bringing this paper to its final form. We are indebted to the anonymous reviewers who provided a critical view of this work and prompted us to clarify many important issues. Margaret Martonosi, Tae Jun Ham, Erik Hagersten, Vijayanand Nagarajan, and Peter Sewell gave us insightful comments on early drafts. This work is a result of the internship 19981/EE/15 funded by the Fundación Séneca-Agencia de Ciencia y Tecnología de la Región de Murcia under the "Jiménez de la Espada" program for mobility, cooperation and internationalization. This work is jointly supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2015-66972-C5-3-R and the Swedish Research Council (VR) grant no. 621-2012-5332.

REFERENCES

[1] Furat Afram, Hui Zeng, and Kanad Ghose. 2013. A group-commit mechanism for ROB-based processors implementing the X86 ISA. In *19th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 47–58. DOI:http://dx.doi.org/10.1109/HPCA.2013.6522306

[2] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. 2009. GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*. 33–42. DOI:http://dx.doi.org/10.1109/ISPASS.2009.4919636

[3] Mehdi Alipour, Trevor E. Carlson, and Stefanos Kaxiras. 2017. Exploring the performance limits of out-of-order commit. In *Int'l Conf. on Computing Frontiers (CF)*. DOI:http://dx.doi.org/10.1145/3075564.3075581

[4] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An open source manycore research framework. In *21st Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 217–232. DOI:http://dx.doi.org/10.1145/2872362.2872414

[5] Gordon B. Bell and Mikko H. Lipasti. 2004. Deconstructing commit. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*. 68–77. DOI:http://dx.doi.org/10.1109/ISPASS.2004.1291357

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.

[7] Colin Blundell, Milo M. K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: Performance-transparent Memory Ordering in Conventional Multiprocessors. In *36th Int'l Symp. on Computer Architecture (ISCA)*. 233–244. DOI:http://dx.doi.org/10.1145/1555754.1555785

[8] Harold W. Cain and Mikko H. Lipasti. 2004. Memory ordering: a value-based approach. In *31st Int'l Symp. on Computer Architecture (ISCA)*. 90–101. DOI:http://dx.doi.org/10.1109/ISCA.2004.1310766

[9] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim. (TACO)* 11, 3, Article 28 (Aug. 2014), 25 pages. DOI:http://dx.doi.org/10.1145/2629677

[10] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk enforcement of sequential consistency. In *34th Int'l Symp. on Computer Architecture (ISCA)*. 278–289. DOI:http://dx.doi.org/10.1145/1250662.1250697

[11] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. 2009. Rock: A high-performance Sparc CMT processor. *IEEE Micro* 29, 2 (March 2009), 6–16. DOI:http://dx.doi.org/10.1109/MM.2009.34

[12] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. 2004. Out-of-order commit processors. In *10th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 48–59. DOI:http://dx.doi.org/10.1109/HPCA.2004.10008

[13] Adrián Cristal, Oliverio J. Santana, Mateo Valero, and José F. Martínez. 2004. Toward kilo-instruction processors. *ACM Trans. Archit. Code Optim. (TACO)* 1, 4 (Dec. 2004), 389–417. DOI:http://dx.doi.org/10.1145/1044823.1044825

[14] Yuelu Duan, David Koufaty, and Josep Torrellas. 2016. SCsafe: Logging sequential consistency violations continuously and precisely. In *Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 249–260. DOI:http://dx.doi.org/10.1109/HPCA.2016.7446069

[15] Nam Duong and Alex V. Veidenbaum. 2013. Compiler-assisted, selective out-of-order commit. *IEEE Computer Architecture Letters* 12, 1 (Jan. 2013), 21–24. DOI:http://dx.doi.org/10.1109/L-CA.2012.8

[16] Marco Elver and Vijay Nagarajan. 2014. TSO-CC: Consistency directed cache coherence for TSO. In *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 165–176.

[17] Ricardo Fernández-Pascual, Alberto Ros, and Manuel E. Acacio. 2017. To be silent or not: On the impact of evictions of clean data in cache-coherent multicores. *Journal of Supercomputing (SUPE)* (March 2017), 1–16. DOI:http://dx.doi.org/10.1007/s11227-017-2026-6

[18] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *In Proceedings of the 1991 International Conference on Parallel Processing*. 355–364.

[19] Dibakar Gope and Mikko H. Lipasti. 2014. Atomic SC for simple in-order processors. In *20thInt'l Symp. on High-Performance Computer Architecture (HPCA)*. 404–415. DOI:http://dx.doi.org/10.1109/HPCA.2014.6835950

[20] Linley Gwennap. 1994. Digital leads the pack with 21164. In *Microprocessor Report, 8(12)*. 249–260.

[21] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *48th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*. 191–203. DOI:http://dx.doi.org/10.1145/2830772.2830800

[22] A. Hilton and A. Roth. 2010. BOLT: Energy-efficient out-of-order latency-tolerant execution. In *16th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 1–12. DOI:http://dx.doi.org/10.1109/HPCA.2010.5416634

[23] Alvin R. Lebeck and David A. Wood. 1995. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *22nd Int'l Symp. on Computer Architecture (ISCA)*. 48–59. DOI:http://dx.doi.org/10.1145/223982.223995

[24] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. 2012. Efficient sequential consistency via conflict ordering. In *17th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*. 273–286. DOI:http://dx.doi.org/10.1145/2150976.2151006

[25] Salvador Petit Marti, Julio Sahuquillo Borras, Pedro Lopez Rodriguez, Rafael Ubal Tena, and Jose Duato Marin. 2009. A complexity-effective out-of-order retirement microarchitecture. *IEEE Trans. Comput.* 58, 12 (2009), 1626–1639. DOI:http://dx.doi.org/10.1109/TC.2009.95

[26] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News* 33, 4 (Sept. 2005), 92–99.

[27] José F. Martínez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *35th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*. 3–14. DOI:http://dx.doi.org/10.1109/MICRO.2002.1176234

[28] Il Park, Chong Liang Ooi, and T. N. Vijaykumar. 2003. Reducing design complexity of the load/store queue. In *36th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*. 411–422. http://dl.acm.org/citation.cfm?id=956417.956555

[29] Parthasarathy Ranganathan, Vijay S Pai, and Sarita V Adve. 1997. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *9th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*. 199–210. DOI:http://dx.doi.org/10.1145/258492.258512

[30] Alberto Ros and Stefanos Kaxiras. 2016. Racer: TSO consistency via race detection. In *49th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*. 1–13. DOI:http://dx.doi.org/10.1109/MICRO.2016.7783736

[31] Amir Roth. 2005. Store Vulnerability Window (SVW): Re-execution filtering for enhanced load optimization. In *32nd Int'l Symp. on Computer Architecture (ISCA)*. 458–468. DOI:http://dx.doi.org/10.1109/ISCA.2005.48

[32] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. 2016. Splash-3: A properly synchronized benchmark suite for contemporary research. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*. 101–111. DOI:http://dx.doi.org/10.1109/ISPASS.2016.7482078

[33] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. DOI:http://dx.doi.org/10.1145/1785414.1785443

[34] Ashok Singhal, David Broniarczyk, Fred Ceraukis, Jeff Price, Leo Yuan, Chris Cheng, Drew Doblar, Steve Fosth, Nalini Agarwal, Kenneth Harvey, and others. 1996. Gigaplane (TM): A high performance bus for large SMPs. In *Hot Interconnects IV*. 41–52.

[35] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers. DOI:http://dx.doi.org/10.2200/S00346ED1V01Y201104CAC016

[36] SPARC International, Inc. 1992. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[37] Rafael Ubal, Julio Sahuquillo, Salvador Petit, Pedro Lopez, and Jose Duato. 2007. VB-MT: Design issues and performance of the validation buffer microarchitecture for multithreaded processors. In *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 429. DOI:http://dx.doi.org/10.1109/PACT.2007.4336257

[38] Rafael Ubal, Julio Sahuquillo, Salvador Petit, Pedro López, and José Duato. 2008. The impact of out-of-order commit in coarse-grain, fine-grain and simultaneous multithreaded architectures. In *Int'l Parallel and Distributed Processing Symp. (IPDPS)*. 1–11. DOI:http://dx.doi.org/10.1109/IPDPS.2008.4536284

[39] Dana Vantrease, Mikko H. Lipasti, and Nathan Binkert. 2011. Atomic Coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. 132–143. DOI:http://dx.doi.org/10.1109/HPCA.2011.5749723

[40] Xiangyao Yu, Hongzhe Liu, Ethan Zou, and Srinivas Devadas. 2016. Tardis 2.0: Optimized time traveling coherence for relaxed consistency models. In *25th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*. 261–274. DOI:http://dx.doi.org/10.1145/2967938.2967942