“THEME ARTICLE”, “FEATURE ARTICLE”, or “COLUMN” goes here: The theme topic or column/department name goes after the colon.

Non-Speculative Load Reordering in TSO

Stefanos Kaxiras
Uppsala University

Trevor E. Carlson
National University of Singapore

Mehdi Alipour
Uppsala University

Alberto Ros
University of Murcia

Load reordering is important for performance. It allows a core to continue performing accesses to the memory system even when there are older, in program order, unperformed accesses (for example, due to long latency misses). The only known solution to allow such reordering in a strong consistency model such as Total Store Order (TSO) has been to reorder speculatively and squash-and-re-execute if caught. We show, for the first time, that we can do the load reordering *non-speculatively* and leave it to the coherence protocol to handle conflicts. We can do this efficiently (without perceptible hardware or performance cost) and without a deadlocks or livelocks. The important new result is that we can now *irrevocably bind speculative loads*. Our solution 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 another core seeing the reordering.

INTRODUCTION

Two decades ago, Gniady et al. explored the observation that accesses in Sequential Consistency (SC) must only *appear* in program order but do not actually have to be (unless others race with them) and demonstrated that given enough resources, all accesses can be *speculatively* re-ordered.[1] On a conflict, speculation is squashed, state is rolled-back to a safe point, and the squashed accesses are re-executed. The same ability is inherited by TSO. TSO relaxes the store→load order but requires the other three program orders (load→load, store→store, and load→store) to be preserved in memory order.[2] Speculation in this case allows loads to be reordered, which is critical for performance. Without load reordering it would be impossible to have further accesses under a miss and memory-level parallelism would suffer. Conveniently, speculation, checkpointing and rollback mechanisms are readily available in today’s processors to support branch prediction or dependence speculation. It is not surprising that speculation has been

the only known approach to keeping the appearance that the load-load order specified by TSO is preserved, while under the hood it is constantly transgressed.

Today, power and energy efficiency concerns prompt us to re-examine architectural approaches and reduce our reliance on speculation. A well-known approach, but one fraught with difficulty, is to commit out of order in a *non-speculative* manner.[3] Out-of-order commit carries the promise of shrinking the speculative window while providing performance of a much larger out-of-order window. In this case, consistency enforcement is one of the stumbling blocks because of its reliance on long and costly speculation to allow the reordering of loads. Consistency speculation is particularly expensive because it must span at least the latency of a miss to main memory.

Going further in this line of reasoning, very simple, highly-efficient, in-order cores provide very shallow speculation support (for a few cycles), certainly not deep enough to cover consistency speculation. Thus, many architectures with prominent in-order implementations rely on relaxed memory models (e.g., Alpha, ARM) that allow liberal access reordering but put the burden on the software to enforce order with memory fences. Unfortunately, software enforces much more order than necessary as it tries to fend-off any possible race that might ever occur, and not just the races that actually *do occur* at run-time. In other words, lack of speculation takes us to tricky relaxed memory models and conservative static software fencing.

In this context, we propose the first non-speculative solution to support reordering of loads in TSO, one of the strongest memory models and one that is widely used in the real world. In our evaluation we show that we can provide non-speculative load reordering TSO without affecting the performance of out-of-order cores. The implications of our solution, however, span a wider range of architectures, from in-order cores to out-of-order cores with out-of-order commit, and a wider range of consistency models, from TSO to relaxed models.

We show that, preserving the appearance of the load-load order in TSO can be efficiently achieved by the coherence protocol in a non-speculative way. The root cause of the load-load reordering problem in TSO is that a race might happen at a bad time: when a reordering can be observed. A reordering of a younger *performed* load can be observed by a conflicting store simply because an older load *is not yet performed*. If the conflicting store occurred slightly later, after the older load is performed, the reordering could not have been observed. Our solution is simple. The race is detected by the invalidation that reaches the younger reordered load. Instead of squashing and re-executing (something that would assume speculation support) we simply withhold the acknowledgement to the invalidation until the reordering disappears (i.e., until the older load is performed). This prevents the conflicting store from obtaining write permission, thus keeping it tucked away in its store buffer. This is allowed by TSO.

The key question is that why this does not deadlock? The answer is that in TSO loads can bypass stores that sit in a store buffer. Of course, we also need to ensure that loads cannot be blocked by stalled stores via the coherence protocol, the directory, or the MHSRs. For this we employ a simple idea: in a situation when a race happens at a bad time, loads that could be blocked, request and receive uncacheable, tear-off copies of the data. To put it simply: we let the loads see the data they want to see without any engagement of the coherence protocol, directories, or MSHRs. This guarantees that there can be no resource conflicts with stalled stores and both deadlock and livelock are prevented.

Our solution allows reordering of loads and enforces order only in the case of a race, by delaying the conflicting store. The speculative solution squashes work and re-executes (incurring additional traffic) while we simply delay a store in its store buffer for a little longer. In terms of performance and coherence traffic, it turns out that our new non-speculative solution is on par with the speculative solution. The difference is that our approach allows irrevocable binding of reordered loads, which gives a significant advantage for out of order commit or for reordering in in-order cores.

NOW YOU SEE ME

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. 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*.[4]

For simplicity, we will use the following hit-under-miss example to explain the basic mechanism. A simpler example for a single address that does not violate consistency but coherence is discussed by Dubois, Annavaram. and Stenström.[5] Our solution applies equally well on coherence misspeculations.

```
Initially x=0; y=0;

Core 0          Core 1
ld ra,y         st x,1
ld rb,x         st y,1

// TSO does not allow ra==1 and rb==0.
If ld rb,x hits and 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.
```

Example 1. Hit-under-miss reordering

Table 1: Six possible *legal* TSO interleavings and their reordering (reo.) for the code in Example 1. TSO is violated only when **ld y** binds to the new value and **ld x** to the old (interleaving 3, reordered)

	TSO Interleaving	Value y, x
1	ld y → ld x → st x → st y	old, old
reo.	ld x → ld y → st x → st y	old, old
2	ld y → st x → ld x → st y	old, new
reo.	ld x → st x → ld y → st y	old, old
3	ld y → st x → st y → ld x	old, new
reo.	ld x → st x → st y → ld y	new, old (illegal in TSO)
4	st x → ld y → st y → ld x	old, new
reo.	st x → ld x → st y → ld y	new, new
5	st x → ld y → ld x → st y	old, new
reo.	st x → ld x → ld y → st y	old, new
6	st x → st y → ld y → ld x	new, new
reo.	st x → st y → ld x → ld y	new, new

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*.

More precisely, a reordering can be observed when 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 Example 1, where one core reads (loads) and another writes (stores) the same two variables but in the opposite order. Table 1 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 interleavings 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 interleavings 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*** (i.e., interleaving 3 with the loads swapped), This yields the combination of values {*new*, *old*} 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.

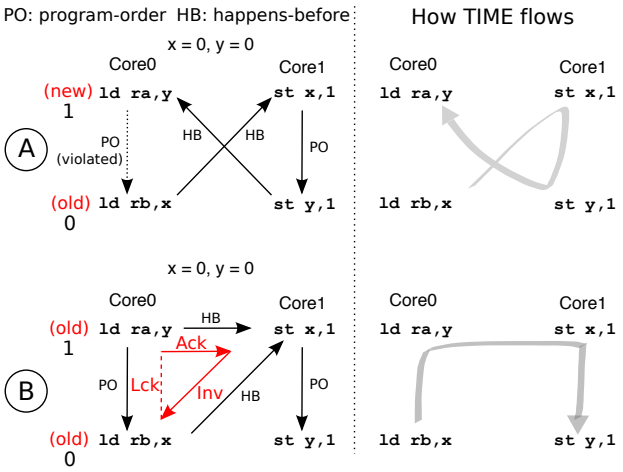

Figure 1: (A) Irrevocably binding the reordered **ld x** can violate TSO if **ld y** sees the new value from **st y**. (B) **ld x** must first see an invalidation from **st x**; delaying the **Ack** of the invalidation (**Inv**) via a "lockdown," (**Lck**) forces **ld y** to *happen before* **st x**, and **st y**.

If we irrevocably bind ***ld x***, our only choice to maintain TSO, according to Table 1, 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***. 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 (i.e., interleaving 3 with the loads swapped turns into interleaving 1 with the loads swapped) and their reordering does not matter. We explain the *lockdown* in the next section.

There are three 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. (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 the size of the load queue minus one.)

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.

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***.

IMPLEMENTATION

The key idea of our approach is 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 all older loads are performed. We propose a deadlock–*free* and livelock–*free* implementation of this concept with three components: i) a *lockdown* mechanism in the cores; ii) a new directory state called *WritersBlock*; and iii) limited use of *uncacheable, tear-off* data for deadlock and livelock prevention.

Lockdown Mechanism: An M-speculative load, i.e., a load that is performed out-of-order with respect to any older load in the same core, *will not acknowledge invalidations* until all previous loads perform. In other words, if an M-speculative load is matched by an invalidation, it will return the acknowledgement only when it becomes *ordered* with respect to older loads. This blocks a conflicting store (which caused the invalidation) in its store buffer. The lockdown mechanism requires minimal changes (just one additional bit per entry) in an ordinary Load Queue of a core. Simple management of this bit ensures that when there are more than one lockdown loads that are matched by an invalidation, only the youngest one returns the acknowledgement when it becomes ordered. Alternatively, lockdowns can be easily implemented in a separate small table for out-of-order commit architectures that remove performed loads from the Load Queue.[6]

WritersBlock: A lockdown by itself is not enough for our purpose. In addition to the lockdown, we need to ensure that:

- A store is blocked (not made globally visible) until *all* existing lockdowns for the store's cacheline address, on all cores, are lifted;
- No further writes for the address in question can take place in memory order before the blocked store is allowed to be performed;
- Loads are never blocked, so that the lockdowns can be lifted to unblock the store.

To achieve these goals we introduce a new *transient* directory state called *WritersBlock*. 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. A WritersBlock state blocks a coherent write request (and all subsequent writes) from completing until the relevant

lockdowns are lifted, yet at the same time, never blocks read requests from accessing the current value of the data.

Uncacheable, tear-off data: Finally, to prevent deadlocks and livelocks, we rely on uncacheable, *tear-off* copies of the data (a tear-off copy is not registered in the directory).[7] The use of uncacheable copies is limited because it is only enabled when the corresponding directory entry enters the WritersBlock state, which is rare. In the following sections we describe our approach to livelock and deadlock under this light.

Livelock

A read request that encounters a WritersBlock state obtains an uncacheable copy of the data. Since it is impossible to set a lockdown on uncacheable data —there is no invalidation ***Ack*** to withhold— such data cannot be used by *younger* loads in the presence of older unperformed loads. Since a younger load cannot be performed with data received from Writersblock, it cannot go into lockdown. This prevents livelock, i.e., an endless stream of new lockdowns indefinitely blocking a store. Thus, an invariant of our approach is the following:

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.

Deadlock

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 (i.e., all previous loads have been performed). The key to understanding how deadlocks can be avoided is the following observation:

The ability of any M-speculative load to become ordered hinges solely on the ability of the oldest unperformed load in the same core to be performed—whichever such load might be at any point in time.

We call such a load the *source of speculation* (*SoS*). If the SoS load is blocked because of a WritersBlock, a deadlock ensues. We need to guarantee that a SoS load cannot be blocked anywhere in the memory system.

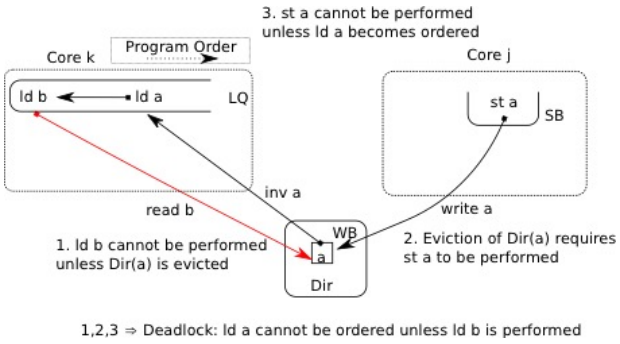

Figure 2: Resource-Conflict Deadlock (Directory Eviction)

We demonstrate our approach with a deadlock example. WritersBlock blocks writes but allows 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 2 shows this case: a reader core (core ***k***) and a writer core (core ***j***) deadlock because of the attempted eviction of the WritersBlock directory entry. The store in core ***j*** is blocked at the directory in the WritersBlock state because it tries to invalidate an M-speculative load in core ***k, ld a***. The store can only proceed if the M-speculative load becomes ordered which will happen when the older load in core ***k***, ***ld b***, is performed. However, this is not possible as ***ld b*** needs the directory entry ***Dir(a)*** held by the blocked store. This is a deadlock.

At this point, 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. The same strategy resolves all other similar resource conflicts in the system.[6]

Atomics, Fences, and Evictions

An atomic RMW instruction, i.e., an atomic load-store pair, represents a special case for consistency models and out-of-order architectures.[2] Because the store of the atomic load-store pair can block, the load of an atomic RMW violates the basic premise of our approach, that SoS loads cannot be blocked. This means that no load following an atomic instruction in the core's instruction window can go into lockdown mode until the atomic instruction is committed. Similarly, no lockdown is allowed to follow a fence instruction until the fence is committed.

With speculative load reordering in TSO, evictions must squash any matching M-speculative load 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. Conservatively, an eviction squashes M-speculative loads in the off-chance that a write would occur in the reordering window. We solve this problem by converting these evictions to silent evictions that do not remove the sharing information from the directory.[6]

CASE STUDY & EVALUATION: OUT-OF-ORDER COMMIT

Our motivation for non-speculative load-load reordering in TSO is the potential for irrevocable binding of loads. We demonstrate this potential, on a *non-speculative* out-of-order commit microarchitecture based on the work of Bell and Lipasti.[3]

We evaluate applications from SPLASH-3 and PARSEC on a x86-like in-house out-of-order processor model that provides TSO and supports out-of-order commit. Three different processors have been simulated, from an efficient Silvermont-class (SLM-class) processor to higher-performing Nehalem-class (NHM-class) and Haswell-class (HSW-class) processors. The WritersBlock cache coherence protocol is modeled in the SLICC infrastructure included in GEMS. The processor simulated consists of 16 cores.

We will first focus on the implications of WritersBlock without considering its out-of-order commit advantages (Figures 3.a, 3.b, 3.c, and 3.d). WritersBlock guarantees that M-speculative loads are never squashed by delaying writes when they are going to invalidate the data of an M-speculative load. If writes are constantly delayed, the protocol performance can drop. Figure 3.a shows that the chances of delaying writes increases with the aggressiveness of the processor (i.e., larger LQ). However, their ratio is still very low (less that 1 write blocked per thousand store operations). WritersBlock issues uncacheable reads to avoid livelocks and deadlocks. It is also important to keep these reads low to keep a low cache miss ratio. We see just 10 uncacheable reads per million loads (Mloads) on average for the largest processors. Therefore, write delays are minimal, even with aggressive cores, as the correspondingly large store buffers can tolerate the small additional latency when prefetching the write permission for the block. The resulting execution time overhead (Figure 3.c) and network traffic overhead are minimal (Figure 3.d).

Overall, these results show that for a typical in-order commit core, WritersBlock is equivalent in performance to the squash-and-re-execute solution. However, for out-of-order commit there is a significant advantage for the non-speculative solution. While the base case must wait in every load reordering for the older loads to perform (regardless of whether there is a conflict or not!), we can commit all reordered loads as long as this is allowed by the other rules for safe (non-

speculative) out-of-order commit.[3] The performance difference (Figure 3.e) for out-of-order commit is over 11% (up to 40% for bodytrack, not shown).

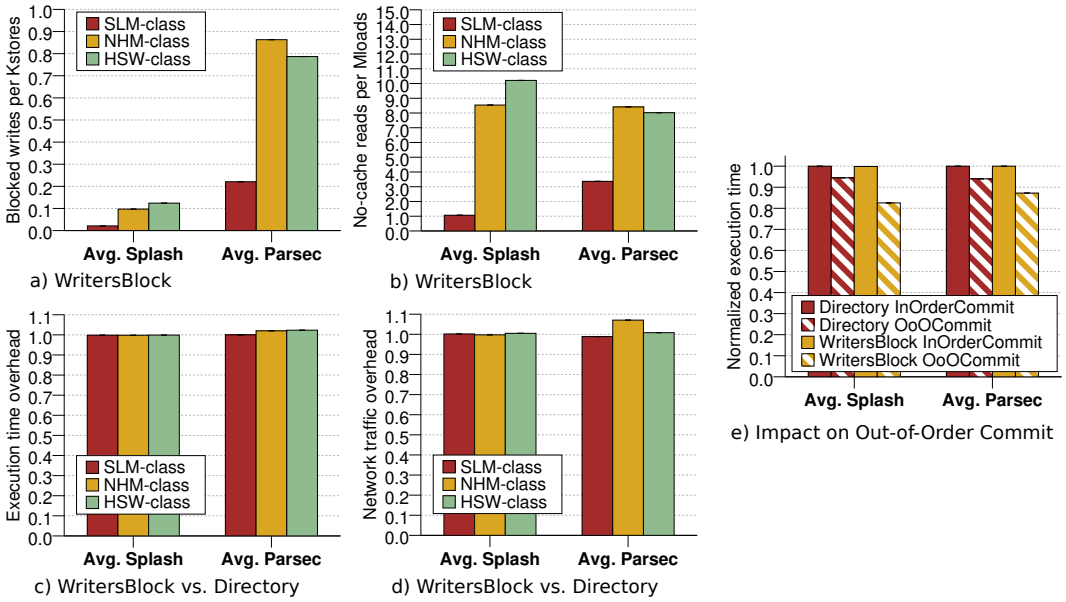

Figure 3. Evaluation results

CONCLUSION

Delaying a conflicting store in its store buffer, until some condition is met, solves an important problem seen in many past and future proposals. In particular, our solution provides, for the first time, *reordering in TSO* minus *the speculation* to:

- in-order cores that continue executing and issuing memory accesses after a miss *without providing* the corresponding speculation depth; in the past, this meant that only a relaxed memory model was a viable option;[8]
- *execute-ahead* approaches such as the UltraSparc Rock which was a checkpoint-based processor,[9] or the recently proposed Load-Slice Core (LSC) that lacks sufficient speculation depth for reordering;[10]
- *accelerators* and *decoupled access-execute* architectures, as for example the recently proposed DeSC architecture, that need to commit loads out-of-order but cannot easily rollback and re-execute on conflicts;[11]
- software (compiler) approaches such as the SoftWare Out-of-Order Processing (SWOOP) core that relies on the compiler to reorder loads and dynamically provide useful work under a miss but would be unrealistic to checkpoint and rollback in software;[12]
- and finally, out-of-order cores with *non-speculative out-of-order commit.*[3,6]

More broadly, our approach removes the enforcement of consistency as one of the major roadblocks for the *irrevocable binding of loads*. This benefit is applicable across the spectrum of core architectures from in-order-execution to out-of-order-execution.

Looking forward, this approach has the potential to affect the implementation of various memory models: load-ordering fences in relaxed memory models become unnecessary as we can now simply guarantee load ordering in the presence of data races while preserving the liberal reordering of the relaxed model in all other situations.

REFERENCES

1. Gniady, Chris, Babak Falsafi, and Terani N. Vijaykumar. "Is sc+ ilp= rc?." *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*. IEEE, 1999.
2. Sorin, Daniel J., Mark D. Hill, and David A. Wood. "A primer on memory consistency and cache coherence." Synthesis Lectures on Computer Architecture 6.3 (2011): 1-212.
3. Bell, Gordon B., and Mikko H. Lipasti. "Deconstructing commit." *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE, 2004.
4. Duan, Yuelu, David Koufaty, and Josep Torrellas. "SCsafe: Logging sequential consistency violations continuously and precisely." High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on. IEEE, 2016.
5. Dubois, Michel, Murali Annavaram, and Per Stenström. Parallel computer organization and design. Cambridge University Press, 2012.
6. Ros, A., Carlson, T. E., Alipour, M., & Kaxiras, S. Non-speculative load-load reordering in tso. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (pp. 187-200). ACM.
7. Lebeck, Alvin R., and David A. Wood. "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors." *ACM SIGARCH Computer Architecture News*. Vol. 23. No. 2. ACM, 1995.
8. Linley Gwennap. 1994. Digital leads the pack with 21164. In Microprocessor Report, 8(12). 249–260.
9. Chaudhry, Shailender, et al. "Rock: A high-performance sparc cmt processor." 29.2 (2009).
10. Carlson, Trevor E., et al. "The load slice core microarchitecture." *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 2015.
11. Ham, Tae Jun, Juan L. Aragón, and Margaret Martonosi. "DeSC: Decoupled supply-compute communication management for heterogeneous architectures." Proceedings of the 48th International Symposium on Microarchitecture. ACM, 2015.
12. Tran Kim-Anh, Alexandra Jimborean, Trevor E. Carlson, Konstantinos Koukos, Magnus Sjalander, Stefanos Kaxiras, "SWOOP: Software-Hardware Co-Design for Non-Speculative, Execute-Ahead, In-Order Cores." To appear ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018).

ACKNOWLEDGEMENTS

We wish to thank Daniel Sorin for his invaluable help. 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.