Speculative Enforcement of Store Atomicity

Alberto Ros
Computer Engineering Department
University of Murcia
Murcia, Spain
aros@ditec.um.es

Stefanos Kaxiras
Department of Information Technology
Uppsala University
Uppsala, Sweden
stefanos.kaxiras@it.uu.se

***Abstract*—Various memory consistency model implementations (e.g., x86, SPARC) willfully allow a core to see its own stores while they are *in limbo*, i.e., executed (and perhaps retired) but not yet inserted in memory order. This is known as store-to-load forwarding and it is a necessity to safeguard the local thread's sequential program semantics while achieving high performance. However, this can lead to counter-intuitive behaviours, requiring fences to prevent such behaviours when needed.**

Other vendors (e.g., IBM 370 and the z/Architecture series) opt for enforcing what we call in this work *store atomicity*, that is, disallowing a core to see its own stores before they are written to memory, trading off performance for a more intuitive memory model. Ideally, we want a stricter model to ease programability at the same time that architects can provide high-performance solutions. We make a simple observation. What holds for any other rule in a consistency model, also holds for store atomicity: it is not a crime to break the rule, unless we get caught.

In this work, we detail the different ways of detecting a store atomicity violation. This leads us to a new insight: a load performed by a forwarding from an *in-limbo* store is not speculative; younger loads performed after that forwarding *are*. Based on this insight we propose an effective and cheap speculative approach to dynamically enforce store atomicity only when the detection of its violation actually occurs. In practice, these cases are rare during the execution of a program. In all other cases (the bulk of the execution of a program) store-to-load forwarding can be done without violating store atomicity. The end result is that we provide the best of both worlds: a more intuitive store-atomic memory model, i.e., the 370 model, with the performance and cost approaching (at an average of just 2.5% and 2.7% overhead for parallel and sequential applications, respectively) that of a non-store-atomic model, i.e., the x86 model.

***Index Terms*—Memory consistency model, store atomicity, multi-copy atomicity, load-to-store forwarding**

I. Introduction

Memory consistency models allow us to reason about program correctness in terms of *program order*, the order in which memory access instructions appear in each thread, and *memory order*, the order in which accesses from different cores read and write memory. Take for example Sequential Consistency (SC) [24], which is widely considered to be the most intuitive memory model for programmers. SC requires that the four possible program orders among loads and stores (load→load, store→store, store→load, load→store) *appear* to respect at all times memory order.

Total Store Order (TSO) relaxes the store→load order with the express purpose of accommodating a *store buffer*. The store buffer is a critical component for performance. It allows a core to retire its store instructions and continue executing without having to wait for the stores to write memory. In TSO, a younger load bypasses older unperformed stores (on different addresses) in the store buffer, hence the store→load order is relaxed. Relaxing just the store→load strikes a good balance between performance (hiding store latency) and clean semantics (the memory model remaining fairly intuitive for the majority of programming idioms) [38].

However, TSO implementations can come in different flavors, as memory models are not defined just by program order guarantees [26]. These flavors differentiate in store atomicity guarantees. We say that *store atomicity* is guaranteed if all cores agree in the memory order of stores (Section II-B).

For example, IBM 370 systems [18], [22] and their present-day descendants, the z/Architecture series [23] opt for respecting store atomicity. We refer to this store-atomic TSO memory model as the 370 model. In the IBM 370, store atomicity is achieved by requiring that a store *in limbo*, in the store buffer, must first be inserted in memory order before it can be forwarded to any local load [22]. This means that whenever a load matches a store in the store buffer, the load is not performed until the store buffer is drained in the memory system (at least up to the matched store). The penalty can be expensive when loads depend frequently on previous stores but store atomicity violations rarely occur in practice. The alternative is to resort to speculation, but the only established approach for such speculation stems from *SC speculation* [9], [19], [20], [30], which as we show in this work, is broader than what is needed.

On the other hand, x86 and SPARC go a step further. They relax *store atomicity* by allowing a core to see its own stores while they are *in limbo*, i.e., *executed (and perhaps retired) but not yet inserted in memory order* [36]. Letting a load take its value from the most recent *matching* store in the store buffer (if such a store exists), known as store-to-load forwarding, is a necessity to safeguard the local thread's sequential program semantics while achieving high performance. However, if this is allowed without guaranteeing that all other threads can also see the same store at that time —i.e., without guaranteeing

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 819134), the European joint Effort toward a Highly Productive Programming Environment for Heterogeneous Exascale Computing (EPEEC) (grant No 801051), and Vetenskapsrdet project 2018-05254.

store atomicity— the resulting memory model assumes less intuitive semantics.

The reason for abandoning store atomicity is threefold: i) it gives a significant performance advantage: it allows us to forward a store value to a load long before the store is ordered in the memory system —after all, satisfying loads as soon as possible is of paramount importance for performance; ii) there is no established solution to enforce store atomicity without incurring significant performance or hardware cost; and iii) *the problem can be delegated to the software. Software memory fencing* is required in cases where the breaking of store atomicity is possible but unacceptable. However, this means that the burden falls on the programmer or compiler to assess whether the code is problematic and properly fence it, for the rare case when a store atomicity violation (during runtime) would result in an unacceptable program behavior.

In this paper, we offer the following insight. Store atomicity is like any other rule in consistency models: *it does not matter if no one is looking*. In other words, we only have to ensure that the rule is enforced if the detection of its violation is imminent. Key for exploiting this insight is to identify under which conditions store atomicity can be *seen violated*. Once we identify these conditions we can *react dynamically* using the processor's built-in speculation and rollback capabilities as prior work has proposed for *other* consistency rules [9], [19], [20], [30]. To help identifying non-store-atomic behaviours in x86 we developed a tool that compares the outcome of a program under the 370 model and the x86 model.[1]

As far as we know, this is the first work that proposes a specific model for *in-window* store-atomicity speculation, distinct from the more general SC speculation. We propose a highly-efficient, *dynamic* enforcement of store atomicity to provide programmers with the illusion of a store-atomic memory model but without its cost. In particular, this work offers the following contributions:

- We demonstrate that *lack* of store atomicity matters in the cases when it can be *observed* and detail the conditions under which such observation can occur (Section III).
- We show that, against the prevailing view [19], [20], [30], loads seeing in-limbo store values are not speculative, but are the *sources of speculation*, i.e., what makes younger loads speculative (Section IV-A).
- We propose an efficient mechanism for store atomicity using just low-overhead in-window speculation (Section IV-B).
- We compare our proposal to three state-of-the-art implementations (Section V and Section VI): x86 (non-store-atomic); 370 (non-speculative store-atomic); and a speculative 370 version directly adapted from state-of-the-art *in-window* SC speculation. As far as we know, this is the first quantitative comparison of these implementations on the same *out-of-order execution* baseline. Our results show that blanket enforcement of store atomicity incurs a heavy cost in execution time (1.27× and 1.23× for parallel and sequential applications, respectively), and that existing speculative solutions are still inneficient. Our approach improves existing speculative techniques by 3.7% and 10.3%, on average, for parallel and sequential applications, respectively, achieving similar performance than x86 while providing a stronger memory model.

[1] https://github.com/alberto-ros/ConsistencyChecker

II. Background

A. The Store Queue and the Store Buffer

In out-of-order cores that relax the store→load order there are actually two structures that are involved: the store queue (SQ) and the store buffer (SB). Stores in the SB are retired but not yet inserted in memory order. Stores in the SQ are at an even earlier stage: they have not been retired, i.e., they are still in the instruction window of out-of-order execution, and hence in the reorder buffer (ROB). In actual implementations (e.g., in Intel x86 architectures) the SQ and the SB are a single physical structure and the division between them is a pointer that separates the retired from the non-retired stores. A load must search both the SQ (for older-in-program-order stores) and the SB to find the most recent store (if it exists in there) to the same address. This is a requirement to satisfy sequential execution semantics. All our discussions apply equally well to retired stores in the SB and non-retired stores in the SQ. To avoid repeating this inconsequential distinction between stores in the SB and stores in the SQ, in the rest of the paper we refer simply to *stores in the store buffer (SQ/SB)*.

B. Store Atomicity, Write Atomicity, and Multi-Copy Atomicity

There is some ambiguity in the literature surrounding the use of the terms "*store atomicity*," "*write atomicity*,", and "*multi-copy atomicity*," often employed to refer to the same property [38], [41]. Here, we give our interpretation of these terms in relation to previous definitions.

The term multi-copy atomicity (MCA) was first defined by Collier [13], and implies that all cores see the new value of a store at the same time. Allowing the local core to see the value of the store before other cores was named by Adve and Gharachorloo [1] as *read-own-write-early*, which led to Trippel et al. [40] to use the term read-own-write-early multiple-copy atomic (rMCA) to name that relaxation of MCA. Wickerson et al. [41] use the term MCA to refer to rMCA.

In our view, *write atomicity* refers to how coherence treats write requests in the memory system, i.e., if a write is atomic or not. For example, a typical invalidation-based MESI protocol that acknowledges a write only after all invalidations have been performed is write-atomic. In contrast, the DASH protocol [25] is not write-atomic as it overlaps the invalidations with the write. So our view of write atomicity matches the definition of rMCA.

Differently, we see *store atomicity* equivalent to MCA, and we define it as follows: *all cores (threads) in the system, without exception, see a store inserted in the global memory order at the same time*. In other words, a core is not allowed to see its own stores before they are globally ordered. Table I

TABLE I
ATOMICITY OF STORE OPERATIONS

Model	Adve & Gharachorloo	Trippel et al.	Ros & Kaxiras
370		MCA	Store atomicity
x86	Read own write early	rMCA	Write atomicity
PC	Read others' write early	non-MCA	Non write-atomic

summarizes this discussion and shows which of the memory models described next represents each category.

C. IBM 370

IBM 370 implements a store-atomic TSO model [18], [22] in which the forwarding of a store to a load is not allowed unless the store is inserted in memory order, i.e., made globally visible to all other cores in the system. This means that when a load matches a store in the store buffer we must wait until that particular store is written to the L1 before we give its value to the load. In this paper, we quantify the cost of an IBM 370-style enforcement of store atomicity.

D. The x86-TSO model

x86 systems implement the x86-TSO model [36], which allows a load to be performed with a store-to-load forwarding from the store buffer. If the load is at the head of the ROB it can retire. This allows the next younger load to retire too when it is performed. As we will show, this is at the heart of the problem with respect to store atomicity. In x86 a core can see only its own stores earlier than other cores. Thus, x86 is a write-atomic or rMCA model.

E. Processor Consistency

Processor Consistency (PC) is defined by Goodman [21]. As IBM 370 and x86, PC forbids all memory reorderings except store→load. However, PC is a weaker model than x86 in that it allows any store to be seen from some cores earlier than other cores. PC is therefore a non-write-atomic model. The behavior of PC can be seen when cache coherence protocols do not enforce write atomicity.

In this paper, we assume a typical invalidation-based MESI protocol that acknowledges a write only after all invalidations have been performed. Under this assumption, it is not possible for a remote core to see a store at a time when another remote core is not able to see it. Therefore, we do not consider PC in this work. For am in-depth discussion of PC, see Adve's and Gharachorloo's tutorial [1].

F. SC speculation

SC is a store-atomic memory model. Naturally, existing approaches for *SC speculation* also guarantee store atomicity [19], [20], [30]. The invariant in those proposals is that *all* loads that bypass any *unperformed* store are *speculative by definition*. This is a stricter condition than the one we adopt in our work for the store-atomic speculative loads. In our new definition, loads reading values from *in limbo* stores are *not speculative*, but they are the *source of speculation* for younger loads that *are* speculative. In this paper, we also compare our proposal with a straightforward adoption of *in-window* SC speculation to the 370 model, for the purpose of providing store atomicity. Note that in our study we do not consider techniques that require *post-retirement speculation* [9] as they are in a different class of complexity and cost. Our goal is to keep the complexity of the speculative support on par with current practice for an easier adoption of our proposal in commodity processors.

III. x86 NON-STORE-ATOMIC SEMANTICS

Relaxing store atomicity results in less intuitive semantics, as we elaborate in this section. The new behaviors that arise from relaxing store atomicity can be classified in two categories, depending on whether we are observing *ordered* or *independent* stores (independent stores that are not bound by program or synchronization order). More specifically, violating store atomicity can result in:

1) *seeing ordered stores in a different order than their memory order.* A core that forwards a store value from the store buffer to a load (possibly) causes that load to see a newer write in memory order, than the write seen by a younger load (in program order) to a different address [6]. (Section III-A)
2) *disagreeing about the order of independent stores.* Two cores that forward their stores to their loads cannot agree in which order their stores appear in memory order. (Section III-B)

In contrast, we show that none of these behaviors can be present in a store-atomic implementation.

A. Ordered Stores

Consider the code shown in Figure 1, which is known as the *mp* (message passing) litmus test. The code has a parallel structure of two program-ordered loads (`ld x` and `ld y` in Core1) and two program-ordered stores (`st y,1` and `st x,1` in Core2) in the opposite order of the loads.

In Figure 1 and all following figures, we use a notation where we show the value of a memory location *before* and *after* it is written by stores (depicted as diamonds): the old value of the memory location before the store is positioned above the store (top corner) and new value that the store writes is positioned below the store (bottom corner).

Figure 1 shows an execution of the code where r_x==1 (meaning that the register loaded by `ld x` takes a value of 1) and r_y==0. This creates a cycle, as it implies that `st x,1` happened before[2] `ld x` and `ld y` happened before[3] `st y,1`. This cycle implies that program order, and therefore TSO, has not been respected, as the loads in Core1 observe the younger store (`st x,1`) being performed *before* the older store (`st y,1`). All other outcomes ($\{0,0\}$, $\{0,1\}$, and $\{1,1\}$ for r_x and r_y respectively) are legal in TSO.

Yet, we can easily re-create this violation if we allow store-to-load forwarding from the store buffer of Core1. Consider

[2] *read from—rf*—in the terminology of Alglave et al. [3]
[3] *from read—fr*—in the terminology of Alglave et al. [3]

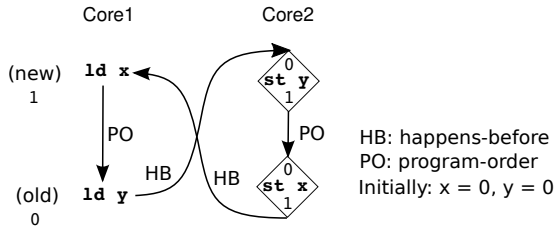

Fig. 1. Forbidden execution in x86 for the *mp* litmus test

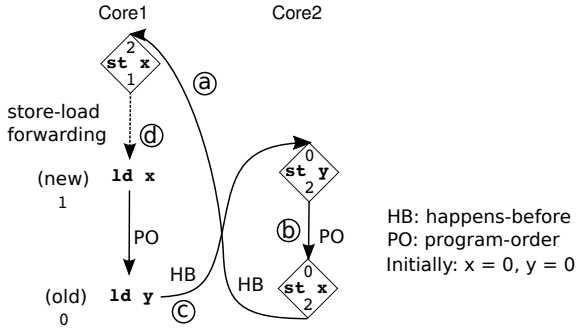

Fig. 2. Allowed execution in x86 but forbidden in store-atomic TSO for the *n6* litmus test

the code shown in Figure 2, which is known as *n6* [36].[4] Compared to the code in Figure 1, n6 adds a new store, `st x,1`, in Core1. The stores in Core2, each, write now a value of 2 in their respective memory locations.

After this code executes in actual x86 implementations[5] we have the following result: r_x==1, r_y==0, `[x]==1`, `[y]==2`, where the notation `[x]==1` means that the memory location `x` has a value of 1 after the code finishes.

Seeing `[x]==1` implies that `st x,1` took place in memory order after `st x,2` (Figure 2 ⓐ).[6] Otherwise, we would see `[x]==2`. Since stores are written to program order by Core2, this implies that *transitively* `st x,1` also took place in memory order after `st y,2` (Figure 2 ⓑ).

The younger load, `ld y`, sees the value `0`. This means that it happened in memory order before[7] `st y,2` (Figure 2 ⓒ). The older load, `ld x`, sees the value `1` as it loads it directly from the store buffer with a store-to-load forwarding[8] from `st x,1` (Figure 2 ⓓ). In Figure 2, if store-to-load forwarding (rfi) enforces memory order, we have a cycle and store atomic behavior. Otherwise, we have a counter-intuitive behavior as there is a dependency that does not follow memory order. As a result, the loads in Core1 see values that contradicts the order of the respective stores of Core2.

Execution Trace (or how did this happen). Delving deeper into how this code produces this behavior we can see the architectural mechanisms that make this possible. The execution trace that leads to this situation is the following. Core1 executes and retires `st x,1`. The store goes into the store buffer so it is *not yet ordered* in the memory system. The store atomicity of `st x,1` is compromised when `ld x` matches it in the store buffer and reads the store's value (store-to-load forwarding). Since `ld x` is now *performed* (it has received its data) and all prior instructions (`st x,1` in this case) have retired, `ld x` is at the head of the reorder buffer and it can also retire.

But what is the actual memory order of `ld x`? It is "hitched" to the store, which —while in the store buffer— is *unordered*. This could be seen as if the load itself remained *unordered*, until the time the store is inserted in the memory order when written to the L1. This observation is key to why loads in Core1 see the stores in Core2 in a different order. In order to restrict this behavior, when `ld y` bypasses `st x,1` in the store buffer and sees the old value of `[y]` (0), it should not have been able to do so until `ld x` is ordered with the insertion of `st x` in memory order.

Furthermore, note that loads seeing different store orders can only happen for loads (e.g., `ld y`) *following* (in program order) a load that receives its value from store-to-load forwarding (i.e., `ld x`). This scenario cannot occur for loads that *precede* (in program order) `ld x` as such loads are inserted before `ld x` in the memory order.

Store Atomicity. In a store-atomic implementation, `ld x` would not be performed until `st x` is inserted in the memory order and all cores can see the new value, thereby preventing loads seeing different store orders, since a cycle will be created in Figure 2 when store-to-load forwarding enforces memory order.

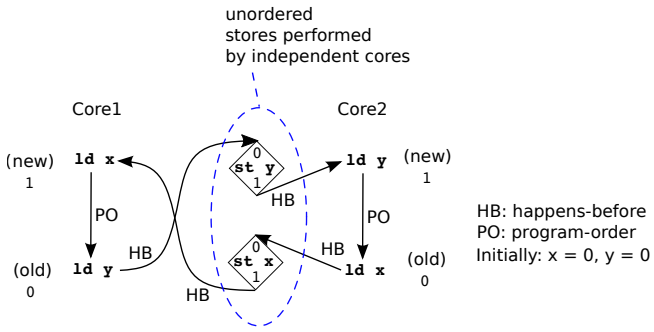

Fig. 3. Forbidden execution in x86 for the *iriw* litmus test

B. Independent Stores

We have shown that two *program-ordered* loads can see two *program-ordered* stores in a different order (Figure 1). The situation is more complex when the stores are not bound by program-order or synchronization-order. Figure 3 shows an example where `st x,1` and `st y,1` are independent (i.e., performed by different cores, without intervening synchronization). This litmus test is known as *iriw* (independent reads of independent writes). There is no predetermined order for

[4] Similar litmus tests are presented by Mador-Haim et al. [27] and by Arvind and Maessen [6].

[5] We used the `litmus7` framework [5] to run litmus tests on Intel Skylake Platinum 8168, Broadwell i5-7400, and Ivy Bridge i5-3230M, and we witnessed this output at a rate of about one in a million.

[6] *write serialization—ws*—in the terminology of Aflglave et al. [3]

[7] *from read*

[8] *read from internal—rfi*— in the terminology of Alglave et al. [3]

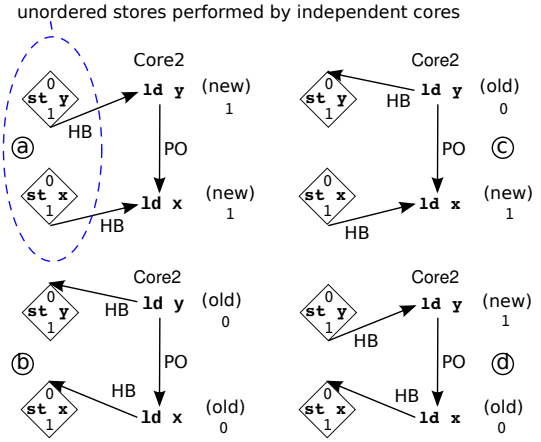

Fig. 4. The four possible outcomes for a core trying to detect whether two independent stores have a particular memory order

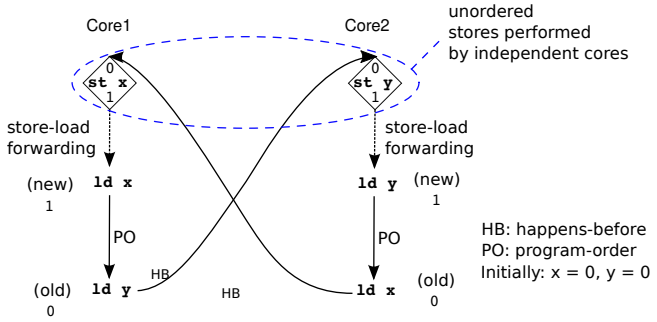

Fig. 5. Store atomicity violation in both cores causes them to not agree on the order of their independent stores

independent stores and perhaps no order can be established (consider for example that the stores can take place simultaneously in separate, independent memory/directory modules).

Let us assume that for the purpose of establishing (if possible) an order for these stores we employ two cores that execute two loads in program order, but each of them in a different order. Core1 executes `ld x` and `ld y` in program order and Core2 executes `ld y` and `ld x` in program order. Core1 can test whether `st x` changes `[x]` *before* `st y` changes `[y]`, but can provide no other ordering information for the two stores. Core2 can only test the opposite case.

Figure 4 shows the four possible outcomes for Core2. If Core2 sees $\{1,1\}$ as the values for `[y]` and `[x]` respectively (Figure 4.ⓐ), it means that each store was performed before its corresponding load but the relative order of the stores cannot be discerned. Similarly, $\{0,0\}$ (Figure 4.ⓑ) means that both stores have not been performed yet and their future order is, of course, unknown. Seeing $\{0,1\}$ (Figure 4.ⓒ) also yields no useful order information for the stores as `ld y` does not see `[y]` change and although the later instruction `ld x` sees `[x]` change, the relative order between the stores remains unknown. It is only when Core2 sees $\{1,0\}$ (Figure 4.ⓓ) that it is certain that `st y` is *before* `st x` in the memory order.

Core2 may not be able to discern any order in the stores. However, if Core2 does detect an order, this order must be respected throughout the whole system. No other core is allowed to see the same memory locations change in an order different than that detected by any other core. Otherwise TSO is violated (Figure 3).

This is the case depicted in Figure 3. Core2 detects that `st y` is actually performed first before `st x` (sees the new value, 1, of `[y]` and the old value, 0, of `[x]`), if Core1 sees `st x` performed first (sees the new value of `[x]` and the old value of `[y]`), a cycle takes place, which means that both cores cannot agree in the order of stores. Actual systems rectify this behavior by squashing one of the loads when the store to the same address performs [19].

While not agreeing in the memory order of two stores results in a non-intuitive behavior, a store-atomicity violation via store-to-load forwarding easily allows us to recreate this situation. Consider the code in Figure 5. In this case we have again two cores trying to detect the order of the stores. The load sequence in one core is designed to detect the opposite store order than the load sequence in the other core. We also need to have the two independent stores in the example. For this, we simply distribute the two stores to the two cores: `st x` in Core1 and `st y` in Core2. The stores are still performed by different cores, hence, they are independent. In x86 systems, this code can result in Core1 seeing `[x]` change before `[y]`, and Core2 insisting on the opposite.[9]

Execution Trace (or how did this happen). The root cause of this behavior is the same as in the case of the ordered stores. In Core1, `ld x` is performed (by the store-to-load forwarding) and since there is no other instruction before it in the reorder buffer it can safely retire. Subsequently, `ld y` bypasses `st x` in the store buffer and sees the old value of `[y]`. In exact symmetry, `ld x` in Core2 is allowed to bypass `st y` in the store buffer and see the old value of `[x]`. No matter how and when the stores of the two cores are written from their respective store buffers in the L1s, each core believes that it has seen the stores in an order that is in direct conflict with the order seen by the other core.

There is no way to tell which core is wrong or right, unless we bring in a third independent core to "*decide*" a memory order for the stores. [10] Even in such a case, it may not be even possible for a third core to discern any order in the stores (e.g., when it can *only* see *both* as either performed or not, as we discussed above).

Store Atomicity. In a store-atomic implementation, `ld x` in Core1 and `ld y` in Core2 would not be able perform until the respective stores in the two cores are written from the store buffers to the L1s and hence inserted in memory order. Exhaustively searching all possible interleavings for the code in Figure 5 reveals that there are only three possible outcomes

[9]We have confirmed this behavior in several recent Intel microarchitectures (Skylake Platinum 8168, Broadwell i5-7400, and Ivy Bridge i5-3230M) using the `litmus7` tools [5].

[10]We note an analogy to the Einstein's thought experiment to explain the special theory of relativity, where two observers traveling at different speeds cannot agree in the simultaneity of two strokes of lightning [17].

TABLE II
ALL POSSIBLE OUTCOMES FOR THE CODE IN FIGURE 5

Case	Core1 [x],[y]	Core2 [x],[y]	Comment
1	1,0 (new,old)	0,1 (old,new)	Disagreement in order
2	1,0 (new,old)	1,1 (new,new)	Core2 cannot see order
3	1,1 (new,new)	1,0 (new,old)	Core1 cannot see order
4	1,1 (new,new)	1,1 (new,new)	None can see any order

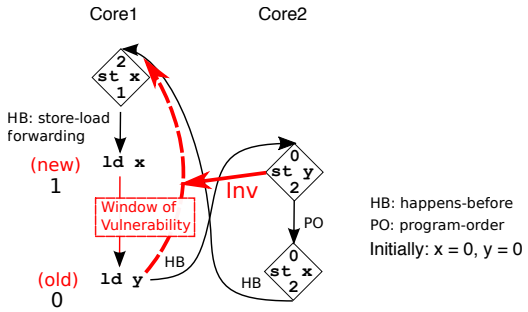

Fig. 6. Window of Vulnerability for invalidations from ordered stores

for a store-atomic implementation. All outcomes are listed in Table II, including the outcome shown in Figure 5 that only appears in the non-store-atomic case (in red).

As we have shown, the first outcome (non-store-atomic) allows the two cores to see the memory locations change in a different order. In the other three outcomes, which are common in store-atomic and non-store atomic implementations, if one of the cores is able to discern an order between the two independent stores (cases 2 and 3) then the other core *cannot* (it sees both stores as performed, hence it cannot tell the order between them); or both cores see both stores performed (case 4), therefore the cores remain agnostic about the store order. This means that *there is no disagreement about the order of independent stores in a store-atomic implementation.*

IV. SPECULATIVE ENFORCEMENT OF STORE ATOMICITY

In the previous section we discussed how store-to-load forwarding in x86 can lead to non-intuitive executions as a consequence of store atomicity violations. The key here is that store-to-load forwarding (seeing a core's own store before other cores do) does not by itself violate the store atomicity of the store. For this to happen, the core in question must also perform at least one more load access to a different memory location that happens to change while the store is still *in limbo* in the store buffer.[11] It is the discrepancy between when we see our own store in relation to an external write and when others observe our store in relation to the same write, that highlights the importance of a store atomicity violation.

This is shown in Figures 6 and 7. A necessary condition for Core1 to break store atomicity in both figures, assuming that `st x` has forwarded the data to `ld x`, is that `st y` in Core2 must be inserted in the memory order sometime between `ld y`

[11]Recall that a store in limbo is executed and (possibly) retired but not inserted in the memory order yet.

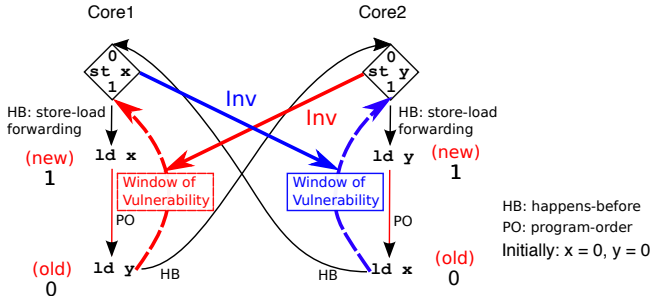

Fig. 7. Window of Vulnerability for invalidations from independent stores

being performed (and seeing the old value of `[y]`) and `st x` in Core 1 being inserted in the memory order. This means that Core1 should receive an invalidation for the memory location `[y]` in the same time window. We call this the *invalidation window of vulnerability* for `ld y`.

More specifically, the reason why store atomicity is violated under x86 in Core1 in a detectable way is that `ld y` is able to retire and exit the reorder buffer before the end of this window (before `st x` is written to the L1). There is nothing in the out-of-order microarchitecture, as we know it today, that can stop `ld y` from retiring and exiting the ROB: `ld x` (which is before `ld y`) is performed from the store buffer and retires, leaving `ld y` at the head of the reorder buffer; at that point, `ld y` is already performed, therefore it can also exit the reorder buffer. An invalidation for `[y]` arriving *after* `ld y` retires but *before* `st x` is written to the L1, seals the fate of the values seeing by the loads, which violates store atomicity.

The only solutions known to guarantee store atomicity are i) to prevent `ld x`, the *consumer* load of the store-to-load forwarding, from being performed until `st x` is inserted in memory order or ii) to treat `ld x` as speculative, and therefore, not allowing it to retire, until the store buffer empties. The first solution, can be a serious impediment to performance as no instruction dependent on `ld x` can progress, but keeps `ld y` speculative for the vulnerability window since there is an older unperformed load in the reorder buffer before it. The second solution also impacts performance as prevents `ld x` from retiring until the store buffer drains. In both cases, within the vulnerability window, if `ld y` is matched by an invalidation, it is squashed as a speculative load, which prevents store atomicity violations. The performance implications of both known techniques are shown in our evaluation.

Evictions. An eviction of cacheline `y`, that occurs during the invalidation window of vulnerability of `ld y`, has the undesirable effect of *filtering out* a possible invalidation, making a violation of store atomicity —in such a case— inevitable. As a *precautionary* measure, to fend off this possibility, evictions are treated the same as invalidations. The same policy is used for speculative load reordering in actual systems [16].

The above analysis, which holds in both cases where store atomicity violation is detected (Section III-A and Section

III-B), forms the basis of our solution. In short, we will make `ld y` in Core1 *speculative* until `st x` is ordered. If an invalidation is received for `[y]` in the interim we must squash `ld y` and re-execute it —in Figure 7, symmetrically, we will do the same (with `x` and `y` switching places) for Core2.

A. A Speculative Solution

There are various ways in which a load can be speculative in an out-of-order core. Previous work has defined loads as:

- *M-Speculative* [14]: the load is performed (receives its data) before a previous *unperformed* load. The oldest unperformed load is called *Source of Speculation* load [31].
- *C-Speculative* [31]: there is an unresolved branch before the load (which means that the load might be on the wrong path).
- *D-Speculative* [31]: there is an unresolved store before the load and the load was issued speculatively assuming no *dependence* with the store.
- *E-Speculative* [34]: there is an unresolved previous instruction that may cause an exception.

We define a new state of speculation for a load:

- *SA-Speculative* (*Store-Atomicity Speculative*): there is an *older* (in program order) load that is performed via a store-to-load forwarding and the corresponding store has not written to cache yet.

To clarify: it is *not* the load performed by store-to-load forwarding that is speculative, but potentially all the loads that follow it. This is a key difference in comparison with the state-of-the-art knowledge regarding speculation. The SA-Speculative state of the load `ld y` holds from the moment it is performed to the moment `st x`, that was involved in the store-to-load forwarding with `ld x` (preceding `ld y`), writes to memory. Our approach can be summarized as follows. *While a load is SA-Speculative: i) it cannot retire, and ii) must be squashed and re-executed if matched by an invalidation or eviction.*

The load that is involved in a store-to-load forwarding is not speculative but it is the *source* of the store-atomicity speculation for all younger loads. We call such a load *SLF load* (Store-to-Load-Forwarded load). To cast a younger load as SA-Speculative, we simply require an "*older*" SLF load. We make no assumptions on:

- *when the store-to-load forwarding takes place* with respect to when the younger load is performed, e.g., the store-to-load forwarding can occur (chronologically) *after* the younger load has been performed.[12]
- *whether the SLF load is still in the reorder buffer* by the time a younger load is performed. The SLF load may have already retired by that time. But even in this case, information about a store-to-load forwarding can be left behind with the store (in the store buffer) and at the load queue, for younger loads to see.

[12]In this case the younger load starts as M-Speculative and becomes SA-Speculative when the older load is performed.

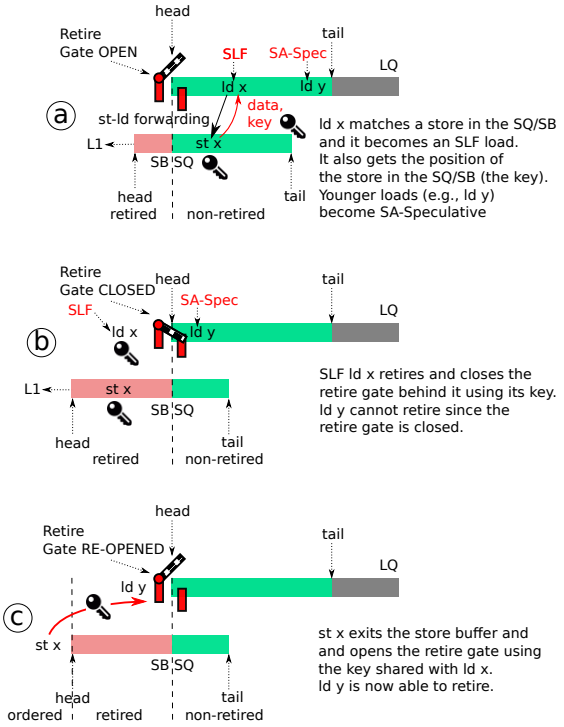

Fig. 8. The basic operation of the Retire Gate.

Keeping track of multiple store-to-load forwardings, the overlapping SA-speculative "shadows" they cast over younger loads, and the points when such shadows are lifted, as stores exit the store buffer, may seem as a daunting task. However, it is not so. We propose a simple and efficient implementation that adds negligible hardware overhead and does not penalize energy or performance.

B. A Simple and Efficient Implementation

Key to a simple implementation are the SLF loads. An SLF load establishes a connection between a store in the store buffer and a "*retire gate*" at the head of the load queue (LQ), thereby preventing the retirement of any younger SA-Speculative loads until the store is inserted in memory order. The SLF load can freely retire (provided that it is not prevented by an even older SLF load). While our solution is based on the LQ, it can be easily adapted to use the ROB instead.

Figure 8 depicts the three parts of our technique:

ⓐ A load receives its value from a store: the load is marked as an SLF load and keeps a pointer to the position of the store in the store buffer. We refer to the value stored in this pointer as the *key*. In other words, the load makes a copy of the key of the store.

ⓑ An SLF load retires: if the load has the valid key of the older store (i.e., the older store has not written yet to the cache), the retire gate is closed by the key of the load, preventing younger SA-speculative loads from retiring.

ⓒ A store exits the store buffer: the retire gate is reopened if the key of the store matches the key that closed the gate.

The next subsections detail the operations on each step.

1) Setting SLF loads on store-to-load forwarding: In current out-of-order processors, when a load executes, it searches the store buffer for a matching store. In case of a hit, the load takes its value from the store. Our proposal extends the implementation by marking at that point the load as being SLF and also by recording the position in the store buffer of that store (i.e., keeps a copy of the key of the store). This entails extending each LQ entry with two new fields. Note that most implementations allow the SLF-load to get the data from a single store buffer entry. In case the underlying implementation allows store-to-load forwarding from multiple store entries, the SLF load copies the key of the younger store providing partial data to the load.

2) Closing the retire gate: The retire gate is nothing more than a single open/closed bit and a register that stores the key. The open/closed bit indicates if the gate is closed (no loads can retire) or open (loads can freely retire). Loads at the head of the LQ (ROB) consult this bit to see if they can retire. The register stores the key that locked the gate when it closed. When the gate is closed it can only be unlocked (and opened) with the same key that it was (closed and) locked. Here is how our mechanism works: When an SLF load retires (assuming the gate is already open), it checks if the store that forwarded the data has already written to cache and has exited the store buffer. In such a case, the load retires without closing the retire gate, as there is no risk of breaking store atomicity (the window of vulnerability of store atomicity has already closed). Otherwise, the SLF load closes the gate behind it and locks the gate using its key. The invariants here are that *there will be one and only one store in the store buffer matching the key* and that *there is only one load that closed the gate*, since the gate is closed after the SLF load retires. Therefore, a simple register can keep the information about the key that closed the gate.

Loads need to check if their matching store has already exited the store buffer. This operation can be implemented in several ways. For example, stores can have a monotonically increasing sequence number (the key), where a lower value indicates an older store [33]. A better implementation, and the one we use in this work, is to augment the *position* of the store in the store buffer with an extra bit per entry (called the *sorting* bit) to account for the wrap-around, as the store buffer is typically a circular buffer [10]. This requires also to augment the key with an extra bit. The retiring load checks the presence of the store by directly accessing the store buffer entry indicated by the position bits of the key, and comparing it sorting bit with the one in the store buffer entry. A match means that the store is still in the store buffer.

3) Reopening the retire gate: The final act takes place when a store exits the store buffer and is written to the L1 cache (Fig. 8.ⓒ). The store compares its key with the key that locked the gate. In case of a match, the store finds that an SLF load has left the LQ closing the retire gate behind it, and that the store forwarded the data to that load since both keys match. After the store writes to cache, and before exiting the store buffer it unlocks and opens the retire gate and invalidates the key stored in the gate register. Younger loads can now retire.

TABLE III
SYSTEM CONFIGURATION

Processor (Skylake-like)	
Issue / Retire width	5 instructions
Reorder buffer	224 entries
Load queue	72 entries
Store queue + store buffer	56 entries
Memory dep. predictor	StoreSet [12]
Branch predictor	L-TAGE [37]
Memory	
Private L1 I&D caches	32KB, 8 ways, 4 hit cycles, pipelined, stride L1 prefetcher [7]
Private L2 cache	128KB, 8 ways, 12 hit cycles
Shared L3 cache (8 banks)	1MB per bank, 8 ways, 35 hit cycles
Directory (8 banks)	8 ways, 200% L2 coverage
Memory access time	160 cycles
Network	
Topology	Fully connected
Data / Control msg size	5 / 1 flits
Switch-to-switch time	6 cycles

C. No Deadlock

In our approach, stores may delay the retirement of *younger* loads. This can cause extra stalls as we will analyze in the evaluation. However, there is no circular dependence in the implementation that would allow a deadlock to occur, since stores cannot be blocked from writing to the L1 cache because of a blocked *younger* load. Once a store retires to the SB it is guaranteed to write to cache and reopen the gate.

D. Storage Requirements

The proposed implementation entails negligible storage overhead. Assuming a 72-entry LQ and a 56-entry SQ/SB (see simulation details in Section V) the extra memory requirements are the following: Each LQ entry is augmented with a SLF bit and the key of the store —$\log_2(56)$+1 bits (for 56 position bits + 1 sorting bit). A total of 8 bits are required per LQ entry. The implementation of the retire gate entails an open/closed bit and a register to store the key of the retired SLF load (7 bits). Finally, the store buffer requires one sorting bit per entry. Overall, the extra memory requirements are just 640 bits (80 bytes).

V. SIMULATION ENVIRONMENT

We simulate a multicore processor consisting of 8 Skylake-like out-of-order cores. We use a detailed in-house out-of-order processor model driven by a Sniper [11]. The out-of-order core implements both macro-operation and micro-operation fusion. The memory hierarchy is modeled with the cycle-accurate GEMS simulator [29], which offers a timing model of the memory hierarchy and the cache coherence protocol. The interconnect is modeled with GARNET [2]. The main architectural parameters of the simulated system are shown in Table III.

We model processors implementing two consistency models found in current systems: 370 and x86. *x86* does not implement any mechanism to enforce store atomicity, and therefore,

is excepted to be the more efficient implementation. The x86 implementation includes support for load-load reordering with in-window speculation, as currently implemented in Intel and AMD processors.

We also evaluate four different processor implementations that provide store atomicity, that is, offer a 370 consistency model: *370-NoSpec*, a store-atomic consistency model with a blanket enforcement of store atomicity as in IBM 370; *370-SLFSpec*, a store-atomic consistency model following SC-like speculation [19], [20], [30] —but strictly for *in-window* speculation— where SLF loads are considered speculative; *370-SLFSoS*, an in-window speculation technique where SLF loads are not considered speculative but the source of speculation (SoS), as described in Section IV-A, and can retire closing the gate behind themselves, reopening the gate when the SB drains; and *370-SLFSoS-key*, our speculative store-atomic implementation where the SLF load acts as a source of speculation and locks the retire gate with the key of its forwarding store, thus reopening the gate as soon as the forwarding store completes the write to L1. We do not model techniques that require *post-retirement* speculation, as they represent a different level of complexity and cost.

We run all applications from the SPLASH-3 [35] and PARSEC 3.0 [8] parallel benchmark suites, with *simsmall* (fmm, ocean_cp, oceanncp, radiosity, radix, raytrace, volrend, water_nsquared, water_spatial, freqmine, streamcluster, swaptions, and vips) and *simmedium* (barnes, cholesky, fft, lu_cb, lu_ncb, blackscholes, bodytrack, canneal, dedup, ferret, fluidanimate, and x264) inputs. Results correspond to their parallel region. Additionally, we present results for sequential applications (the whole SPECrate CPU 2017 benchmark suite [39]) with reference input sets and reporting numbers for the execution of $\approx$ 1 billion instructions after the warm up phase.

VI. Evaluation

A. Impact of store-atomicity speculation

Adding speculative support for store atomicity allows forwarding from non-performed stores to loads, which is fundamental for applications' performance. However, speculation sometimes fails, having to re-execute the instructions again. Until the SA-speculation is validated (forwarding stores are written to L1), SA-speculative loads cannot retire, thus having to wait at the head of the ROB (when the gate is closed) and stalling the processor if its resources (ROB, LQ or SQ) are full. These re-executions and stalls can jeopardize performance if they occur frequently. This section explains why the performance penalty of speculative store atomicity is small, leading to performance that is close to a non-store-atomic model as x86.

Table IV offers a detailed characterization of parallel (top) and sequential (bottom) applications when running under our speculative store atomicity implementation (*370-SLFSoS-key*). The first column shows the evaluated applications. The second column presents the number of retired instructions for each application. The third column shows the percentage of retired loads with respect to retired instructions, averaging around 24% for both parallel and sequential applications. Then we go to the key column that shows that forwarding is not a frequent operation. Only *barnes* has a very high forwarding ratio (18.3% of the total instructions). This is due to the large number of recursive function calls that use the stack to write and read parameters. Specifically, the recursive function "walksub" in *barnes* is by far the most store-load forwarding intensive function of the benchmark. On average, only 3.69% of the instructions are loads that get the data forwarded from a store (SLF loads) for parallel applications. Regarding sequential applications only *500.perlbench_2* and *511.povray* show more than 10% forwarded loads, which is still a low percentage compared to *barnes*. On average, sequential applications incur slightly more forwarding than parallel applications (4.55%).

In most of these cases, the store that forwarded the data to the load writes to memory before the load retires, and therefore, the retire gate is never closed. Only 1.12% of instructions stall at the head of the ROB because the gate is closed (the fifth column) for parallel applications, and this stall takes an average of 18.4 cycles until the gate opens again (the sixth column). For sequential applications, the average of stalls is 1.48% and the number of cycles the gate is closed is lower than for parallel applications (11.5 cycles). As a conclusion, closing the gate is a rare and short-lived event, and does not cause noticeable performance degradation, as we show in the next section.

Finally, the other possible disadvantage of speculation is the occurrence of misspeculations. As we mentioned earlier, store atomicity misspeculations are a rare event. The seventh column in Table IV shows the percentage of instructions that are re-executed due to store atomicity misspeculations, accounting from the speculative load that is caught by an invalidation or replacement to the tail instruction in the ROB. This percentage is 0.492% for parallel applications and 0.565% for sequential applications, so it hardly impacts execution time. Two outliers, in this case, are *x264*, with 10.2%, and *505.mcf*, with 11.7% re-executed instructions due to store atomicity misspeculation. This high re-execution percentage in *x264* is due to the code in the "pthread_cond_wait" function, frequently called in this application. This code incurs store-to-load forwarding in a highly contended synchronization variable. The reason for the large number of misspeculations in *505.mcf* is due to frequent cache evictions that hit SA-speculative loads in the LQ. In general, all misspeculations seen in sequential applications are due to cache evictions.

B. x86, 370, and SC speculation

This section compares the performance of our speculative store atomicity solution to three state-of-the-art implementations: *x86*, 370 without store-atomicity speculation (*370-NoSpec*), and 370 with SC-like speculation (*370-SLFSpec*). We present two configurations for our solution. The first one, *370-SLFSoS* is just presented to show the advantages of letting the SLF load retire. The second one, *370-SLFSoS-key* is our

TABLE IV
CHARACTERIZATION OF STORE-ATOMICITY SPECULATION FOR PARALLEL (TOP) AND SEQUENTIAL (BOTTOM) APPLICATIONS

Benchmark	Instructions	Loads (% total instr.)	Forwarded (% total instr.)	Gate Stalls (% total instr.)	Avg. stall cycles per gate stall	Re-executed instr. (% total instr.)
barnes	2230309927	31.780	18.336	5.929	6.460	0.194
blackscholes	1053954449	19.745	7.272	2.208	4.428	0.001
bodytrack	3871819525	17.915	4.119	1.028	4.375	0.292
canneal	911238793	24.259	2.755	0.730	5.226	0.035
cholesky	873398060	26.320	1.604	0.406	6.188	0.027
dedup	852338767	13.762	6.481	1.467	3.183	0.012
ferret	843881294	20.542	3.527	1.411	11.112	0.147
fft	2305314837	17.282	0.010	0.006	6.113	0.000
fluidanimate	3439523371	25.233	1.044	0.316	8.459	0.035
fmm	1391062359	15.439	0.294	0.118	19.345	0.013
freqmine	2594696106	26.120	2.584	1.185	5.960	0.167
lu_cb	4160074138	22.165	0.230	0.124	4.850	0.015
lu_ncb	4331579576	24.261	1.352	0.636	16.362	0.048
ocean_cp	958925716	30.497	0.031	0.017	94.560	0.002
ocean_ncp	876550467	27.233	0.064	0.033	52.584	0.007
radiosity	1071130503	29.947	4.201	0.628	7.783	0.106
radix	160864073	28.182	1.411	0.790	98.644	0.235
raytrace	1582601968	28.501	5.625	2.045	8.151	0.145
streamcluster	1352721745	29.899	0.031	0.020	53.851	0.000
swaptions	2086529095	24.576	4.498	2.184	5.284	0.245
vips	4360543980	18.061	1.962	0.534	5.000	0.005
volrend	801497112	24.514	5.097	1.353	5.484	0.184
water_nsquared	276836113	26.834	7.687	1.680	6.181	0.145
water_spatial	2259979795	27.851	8.669	1.608	6.292	0.045
x264	1368542748	26.209	3.314	1.432	13.723	10.191
Average	**1840636580**	**24.285**	**3.688**	**1.115**	**18.384**	**0.492**

Benchmark	Instructions	Loads (% total instr.)	Forwarded (% total instr.)	Gate Stalls (% total instr.)	Avg. stall cycles per gate stall	Re-executed instr. (% total instr.)
500.perlbench_1	964505810	23.866	7.527	2.686	6.967	0.146
500.perlbench_2	973276968	29.159	11.192	3.969	4.979	0.038
500.perlbench_3	929430787	7.889	1.075	0.378	4.979	0.020
502.gcc_1	980611000	24.143	8.032	2.094	9.263	1.152
502.gcc_2	980660274	24.132	8.027	2.090	9.293	1.156
502.gcc_3	984563265	24.955	8.300	2.183	9.568	0.987
502.gcc_4	983294223	25.847	8.044	2.188	9.900	1.054
502.gcc_5	983293143	25.847	8.043	2.187	9.896	1.063
503.bwaves_1	973162848	30.147	1.722	0.782	17.455	0.032
503.bwaves_2	973162943	30.147	1.722	0.782	17.450	0.034
503.bwaves_3	1013214128	33.200	2.094	0.814	29.580	0.044
503.bwaves_4	980379698	30.310	1.765	0.855	35.334	0.040
505.mcf	1033168380	29.973	4.958	2.411	13.084	11.722
507.cactuBSSN	988799146	31.857	5.593	1.479	18.801	0.014
508.namd	957464484	23.369	2.448	1.316	3.973	0.008
510.parest	977387085	33.230	1.852	0.530	6.907	0.067
511.povray	1047422921	30.513	10.185	2.911	5.772	0.003
519.lbm	939699615	20.561	7.695	3.074	74.749	0.440
520.omnetpp	1011815225	27.695	7.978	2.437	15.927	0.329
521.wrf	1006331121	25.615	2.004	0.730	11.495	0.016
523.xalancbmk	1036626285	26.679	2.804	0.700	8.810	0.167
525.x264_1	910390076	22.529	3.381	0.607	6.611	0.012
525.x264_2	911740169	23.605	1.397	0.303	8.870	0.015
525.x264_3	909357540	22.722	2.841	0.520	6.546	0.006
526.blender	982134804	23.531	6.116	1.752	5.680	0.139
527.cam4	900052617	22.683	0.001	0.000	0.000	0.000
531.deepsjeng	1005818672	22.159	6.743	2.632	5.926	0.960
538.imagick	901182035	18.552	0.103	0.023	6.798	0.001
541.leela	1013351926	23.706	5.085	2.031	6.795	0.393
544.nab	966696584	22.047	4.176	1.426	5.726	0.126
548.exchange2	1212408138	24.982	4.140	1.289	6.112	0.032
549.fotonik3d	1000196710	20.950	7.703	2.800	6.293	0.012
554.roms	1034743008	25.549	3.700	1.037	10.122	0.016
557.xz_1	925428657	14.427	3.312	1.913	4.493	0.092
557.xz_2	930899613	10.098	1.064	0.181	5.094	0.002
557.xz_3	928391278	12.466	0.981	0.167	5.096	0.002
Average	**979196144**	**24.143**	**4.550**	**1.480**	**11.510**	**0.565**

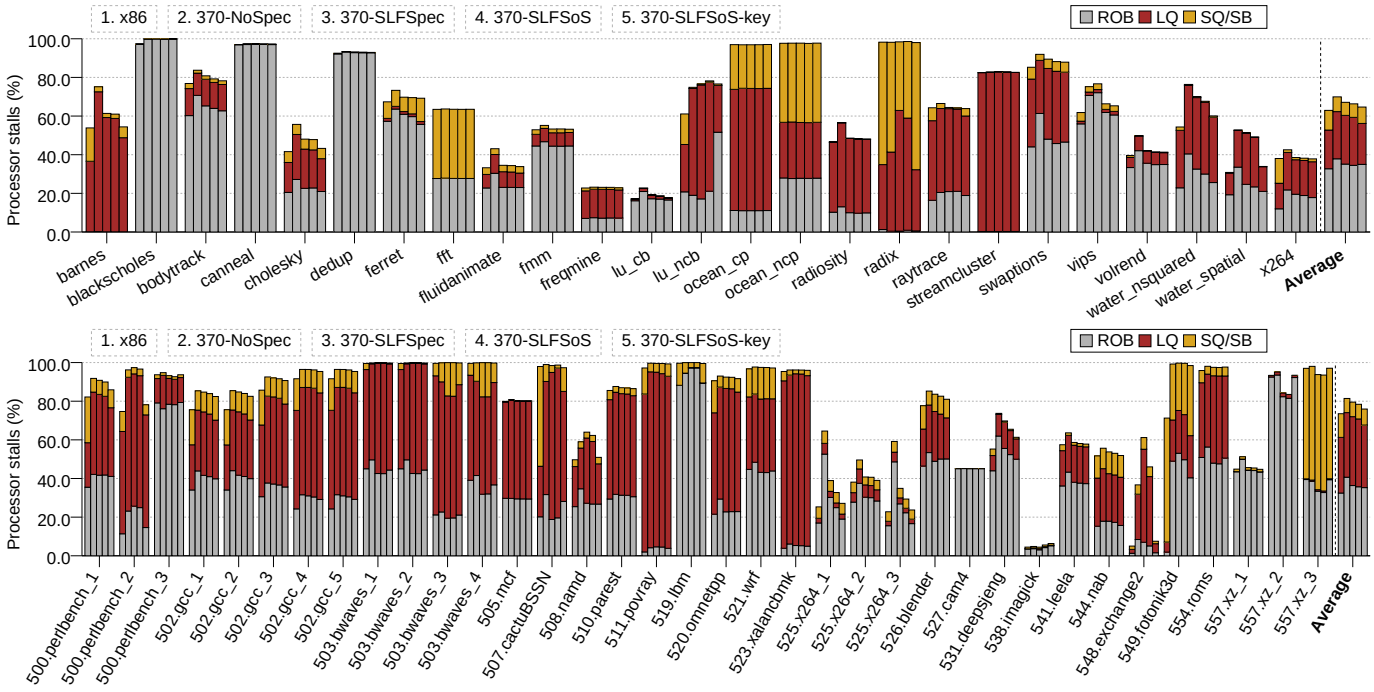

Fig. 9. Percentage of processor stalls for parallel (top) and sequential (bottom) applications

efficient implementation of speculative enforcement of store atomicity.

Figure 9 shows the percentage of cycles in which the processor cannot make progress due to a full ROB, LQ, or SQ/SB for both parallel (top) and sequential (bottom) applications. Preventing store-to-load forwarding until the store writes to memory (*370-NoSpec*) delays the execution of loads, which results in extra processor stalls with respect to *x86* when resources (e.g., ROB, LQ) fill up. *370-SLFSpec* reduces the stalls cycles by allowing SLF loads to perform speculatively. Not being able to retire these loads until the store buffer empties, still entails extra stalls compared to *x86*. Our solutions allow SLF loads to retire, thus reducing ROB/LQ stalls (*370-SLFSoS*) and the use of the key allows the retire gate to open even earlier than in previous speculative solutions (*370-SLFSoS-key*), achieving stall levels close to *x86*.

Figure 10 shows the execution time of all four store-atomic implementations normalized to *x86* and for both parallel (top) and sequential (bottom) applications. The increase in processor stalls translates to increased execution time. Blanket enforcement of store atomicity (*370-NoSpec*) increases execution time over x86 by 1.27x (1.23x for sequential applications), on average. SC-like speculative store atomicity (*370-SLFSpec*) reduces execution time compared to *370-NoSpec* but still entails a noticeable overhead (1.07x for parallel applications and 1.14x for sequential applications). For some applications (e.g. *radix* and *519.lbm*), *370-NoSpec* can outperform *370-SLFSpec* as in *370-NoSpec* as soon as the previous stores calculate their address, the load can execute non speculatively, while speculation in *370-SLFSpec* remains until all previous stores exit the store buffer. Allowing SLF loads to retire (*370-SLFSoS*) brings execution time down to 1.05x (1.12x for sequential applications), and when adding the key mechanism (*370-SLFSoS-key*) the overhead is just 1.025x (1.027x for sequential applications). The outlier here is *radix*, which is dominated by long-latency writes that stress the SQ/SB capacity (Figure 9, top), thus having the larger average stall cycles per gate stall (99 cycles, Table IV). The average performance improvement of our proposal compared to speculative SC is moderate for parallel applications (3.7%) but excels for sequential applications (10.3%).

Finally, we do not significantly alter dynamic energy consumption in the structures involved in our techniques (SQ/SB, LQ, ROB) as we do not require extra snoops in our mechanism, and naturally, overall energy consumption (including static energy) is dominated by the changes in execution time.

VII. Related Work

SC++. One of the reference points for our work is the work of Gniady et al. [20] which shows that with deep enough *post-retirement* speculation, SC can be as liberally relaxed as RC. Since that work aims to cover reordering in SC, it must obviously cover store atomicity, as there can be no SC without it. Gniady et al. introduce a *post-retirement* speculative version of SC, called SC++, which allows any instruction reordering (loads and stores bypassing each other). More specifically, *all* loads that bypass any *unperformed* store are *speculative by definition*. This is the fundamental difference between the work of Gniady et al. and our work. In the memory model that we focus on, *it is perfectly legal for a load to overtake*

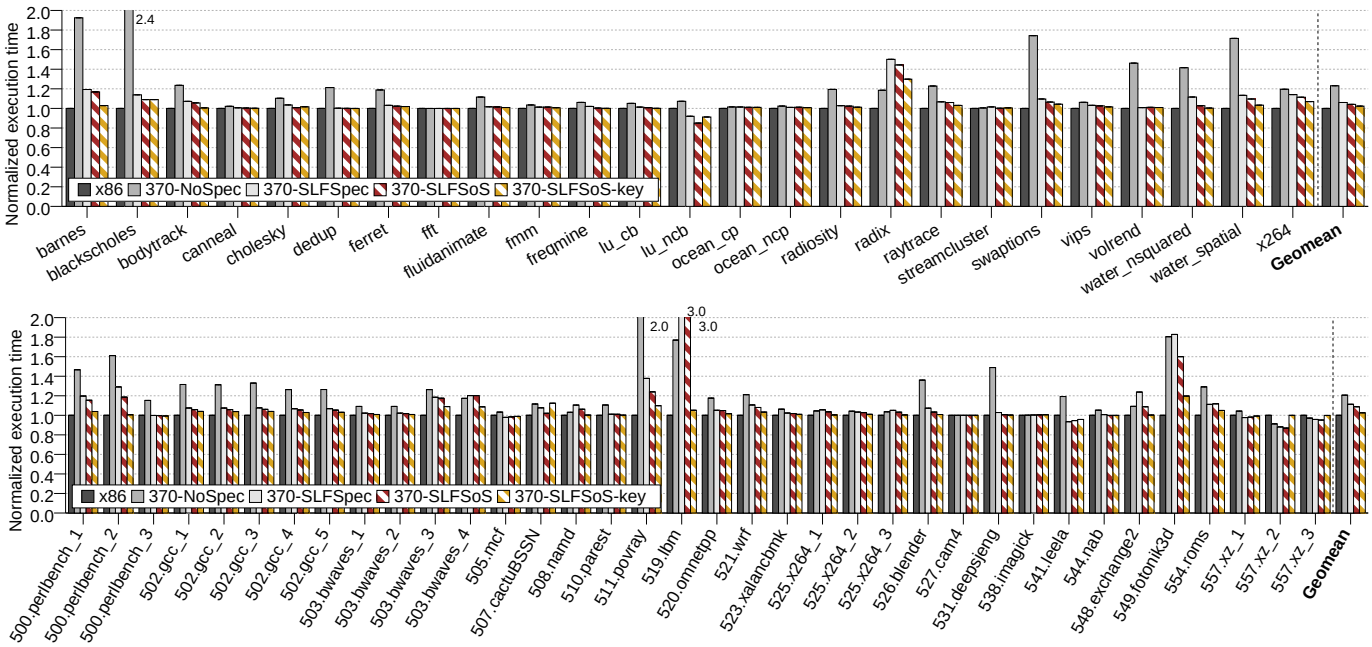

Fig. 10. Normalized execution time for parallel (top) and sequential (bottom) applications

a store and not be speculative. Finally, our proposal strictly concerns *in-window speculation*, in contrast to *post-retirement* speculation, a more complex technique not implemented in current processors [9], [28].

Store Atomicity. Arvind and Maessen show that memory models are effectively defined by the allowable instruction ordering and store atomicity [6]. Differences between the *non-store-atomic* x86 and the *store-atomic* 370 were described in earlier work [18]. Arvind and Maessen describe a single aspect of how a store-atomicity violation becomes evident: ordered loads see ordered stores in a different order. Their intent is only to model this behavior, rather than to address it. In contrast, we demonstrate the precise conditions for store atomicity violations (*window of vulnerability for invalidations*) and propose an efficient solution to dynamically prevent such violations from occurring. As far as we know, this is the first work that describes such a solution.

Speculative fences. An alternative to our speculative enforcement of store-atomicity is to add the necessary fences to the programs and enforce them speculatively. WeeFence [15] is an example of an speculative enforcement of fences. Unfortunately it is not practical for store-atomicity as it requires the target memory block of a forwarded load to be stored in the local cache of the core before executing the load (Section 5.2). In addition, WeeFence involves non-trivial changes in the memory system, while our proposal leaves the memory system unmodified.

Sentinels. The use of our key may resemble the sentinels employed in prior work [10], [32]. In our proposal we leverage the sorting bit proposed by Buyuktosunoglu et al. [10]. Different from NoLQ [32], we do not require heavy modifications or extra snoops in the store buffer, but we reuse the snoop already performed for every load in the store buffer to get a copy of the store *key*.

VIII. Conclusion

Until now, store atomicity represented a *dilemma*. On one hand, some architectures (e.g., x86, SPARC) chose to abandon store atomicity for more performance and delegate any possible problems to be solved with software fencing. On the other hand, some architectures (e.g., IBM 370 and z/Architecture) put more emphasis on a stronger and more intuitive memory model. Speculation for store atomicity, based on SC-speculation, on the other hand, seems unable to fully bridge the performance gap between the two memory models.

Proponents of the former approach (Intel and AMD) may argue that the problems stemming from the lack of store atomicity are rare and, if they appear, the responsibility for correctness falls to patching the software with fences [4]. Proponents of the latter (IBM) may argue that having a more intuitive memory model is of critical importance and some performance loss can be acceptable in exchange of this virtue.

In this work, *we bridge the performance gap* by presenting, a solution to guarantee store atomicity *dynamically, when needed*. It has a small impact on performance compared to a baseline without store atomicity ($\approx$ 2%) and negligible implementation cost (for supporting the concept of SA-Speculation). On the other hand, it offers store atomicity *with a sizable performance increase (10.3% for sequential applications)* over an implementation that relies on existing in-window speculation.

Acknowledgment

We thank the anonymous reviewers for their constructive comments which helped to improve the quality of this work.

REFERENCES

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[3] J. Alglave, "A formal hierarchy of weak memory models," *Formal Methods in System Design (FMSD)*, vol. 41, no. 2, pp. 178–210, Oct. 2012.

[4] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," in *22nd International Conference on Computer Aided Verification (CAV)*, Jul. 2010, pp. 258–272.

[5] ——, "Litmus: Running tests against hardware," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 41–44.

[6] Arvind and J.-W. Maessen, "Memory model = instruction reordering + store atomicity," in *33rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2006, pp. 29–40.

[7] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*, 1st ed. Cambridge University Press, 2009.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[9] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: Performance-transparent memory ordering in conventional multiprocessors," in *36th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2009, pp. 233–244.

[10] A. Buyuktosunoglu, A. El-Moursy, and D. H. Albonesi, "An oldest-first selection logic implementation for non-compacting issue queues," in *15th Annual Int'l ASIC/SOC Conference*, Sep. 2002, pp. 31–35.

[11] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations," in *Conf. on Supercomputing (SC)*, Nov. 2011, pp. 52:1–52:12.

[12] G. Z. Chrysos and J. S. Emer, "Memory dependence prediction using store sets," in *25th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1998, pp. 142–153.

[13] W. W. Collier, *Reasoning About Parallel Architectures*. Prentice-Hall, Inc., 1992.

[14] Y. Duan, D. Koufaty, and J. Torrellas, "Scsafe: Logging sequential consistency violations continuously and precisely," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 249–260.

[15] Y. Duan, A. Muzahid, and J. Torrellas, "WeeFence: Toward making fences free in tso," in *40th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2013, pp. 213–224.

[16] M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*. Cambridge University Press, 2012.

[17] A. Einstein, *Relativity: The Special and General Theory*. Henry Holt and Company, 1920.

[18] K. Gharachorloo, "Memory consistency models for shared-memory multiprocessors," Western Research Laboratory, Research report 95/9, Dec. 1995.

[19] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *20th Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1991, pp. 355–364.

[20] K. Gniady, B. Falsafi, and T. Vijaykumar, "Is SC + ILP = RC?" in *26th Int'l Symp. on Computer Architecture (ISCA)*, May 1999, pp. 162–171.

[21] J. R. Goodman, "Cache consistency and sequential consistency," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep. TR1006, Feb. 1991.

[22] IBM, "System/370 principles of operation," *GA22-7000, IBM Data Processing Division, White Plains, NY.*

[23] ——, "z/architecture principles of operation," *SA22-7832-09, IBM Data Processing Division, White Plains, NY.*

[24] L. Lamport, "Times, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[25] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. A. Horowitz, and M. S. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[26] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures," in *42nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2015, pp. 388–400.

[27] S. Mador-Haim, R. Alur, and M. M. Martin, "Generating litmus tests for contrasting memory consistency models," in *22nd International Conference on Computer Aided Verification (CAV)*, Jul. 2010, pp. 273–287.

[28] S. P. Marti, J. S. Borras, P. L. Rodriguez, R. U. Tena, and J. D. Marin, "A complexity-effective out-of-order retirement microarchitecture," *IEEE Transactions on Computers (TC)*, vol. 58, no. 12, pp. 1626–1639, Jul. 2009.

[29] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[30] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton, "An evaluation of memory consistency models for shared-memory systems with ilp processors," in *7th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 1996, pp. 12–23.

[31] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, "Non-speculative load-load reordering in tso," in *44th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2017, pp. 187–200.

[32] A. Ros and S. Kaxiras, "The superfluous load queue," in *51st IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2018, pp. 95–107.

[33] A. Roth, "Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization," in *32nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2005, pp. 458–468.

[34] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 723–735.

[35] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[36] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[37] A. Seznec, "The L-TAGE branch predictor," *The Journal of Instruction-Level Parallelism*, vol. 9, pp. 1–13, May 2007.

[38] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[39] Standard Performance Evaluation Corporation, "SPEC CPU2017," 2017. [Online]. Available: http://www.spec.org/cpu2017

[40] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory model verification at the trisection of software, hardware, and ISA," in *22nd Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Apr. 2017, pp. 119–133.

[41] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically comparing memory consistency models," in *44th Symp. on Principles of Programming Languages (POPL)*, Jan. 2017, pp. 190–204.