The Superfluous Load Queue

Alberto Ros
Department of Computer Engineering
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*—In an out-of-order core, the load queue (LQ), the store queue (SQ), and the store buffer (SB) are responsible for ensuring: i) correct forwarding of stores to loads and ii) correct ordering among loads (with respect to external stores). The first requirement safeguards the sequential semantics of program execution and applies to both serial and parallel code; the second requirement safeguards the semantics of coherence and consistency (e.g., TSO). In particular, loads search the SQ/SB for the latest value that may have been produced by a store, and stores and invalidations search the LQ to find speculative loads in case they violate uniprocessor or multiprocessor ordering. To meet timing constraints the LQ and SQ/SB system is composed of CAM structures that are frequently searched. This results in high complexity, cost, and significant difficulty to scale, but is the current state of the art.**

Prior research demonstrated the feasibility of a *non-associative* LQ by *replaying* loads at commit. There is a steep cost however: a significant increase in L1 accesses and contention for L1 ports. This is because prior work assumes Sequential Consistency and completely ignores the existence of a SB in the system.

In contrast, we intentionally delay stores in the SB to achieve a *total* management of stores and loads in a core, while still supporting TSO. Our main result is that we eliminate the LQ *without* burdening the L1 with extra accesses. Store forwarding is achieved by delaying our own stores until speculatively issued loads are validated on commit, entirely in-core; TSO load→load ordering is preserved by delaying remote external stores in their SB until our own speculative reordered loads commit. While the latter is inspired by recent work on non-speculative load reordering, our contribution here is to show that this can be accomplished *without having a load queue*. Eliminating the LQ results in both energy savings and performance improvement from the elimination of LQ-induced stalls.

***Index Terms*—Memory Consistency, Out-of-order Execution, Total Store Order**

I. Introduction

In out-of-order cores the load queue (LQ) is essential for issuing loads *speculatively* while guaranteeing correctness. This correctness consists of (1) intra-thread memory dependence and store forwarding; (2) consistency load→load order; and (3) coherence for loads on the same address, independently of the consistency model.

Memory dependence store forwarding: A load must read the latest value written by a store to the same address, if no other core wrote that location in the interim. Waiting for all previous stores to resolve their target address and then issuing subsequent loads can result in a severe performance penalty. High-performance processors issue loads speculatively over older stores with unknown addresses. When a store resolves its address, or when the store is ready to commit, the LQ is searched for younger speculatively-executed loads matching the store address. If there is a match, such loads and subsequent instructions are squashed.

Consistence load→load ordering: Some consistency models such as TSO specify that loads must appear in program order. Waiting for a previous load to complete is time consuming, and processors speculatively execute loads out-of-order [1]. In such cases, the load→load order can be violated if a remote core writes the memory location of such a speculative load. Therefore, upon receiving an invalidation or upon a cache eviction a core searches its LQ. If a match for a speculatively reordered load occurs, the load and all subsequent instructions are squashed.

Coherence load→load ordering (same-address loads): While the previous correctness requirement concerns specific memory models (e.g, TSO) and one could argue limited applicability, there is a case of load reordering that is a *coherence* problem, not a consistence problem. This requirement is therefore *universal* in any coherent system, regardless of the memory model, thus superseding the consistence requirement. More specifically, when a load is reordered over another load on the *same address* (perhaps because the older load did not resolve its address in time) the exact same requirements hold: an invalidation or an eviction must search the LQ and squash any matched speculatively-executed load. Excellent discussions of this phenomenon appear in Dubois, Annavaram, and Stenström [2], Bell and Lipasti [3], and Cain and Lipasti [4].

To summarize: the state of the art today is that processors aggressively execute loads speculatively out-of-order and detect memory dependence violations or memory ordering violations (for consistence *or* coherence) and squash the offending loads if necessary. The LQ has the central role in this approach and is searched on every incoming invalidation, cache eviction, or address resolution of a store.

The LQ must be designed in a way to both keep the order or loads and support multiple, fast, single-cycle, associative searches on address. The result is a complex CAM structure that keeps both FIFO order and allows searching with priority matching (e.g., finding the *oldest* match). Not only the LQ

This work is supported by the Spanish MINECO, European Commission FEDER funds, under grant “TIN2015-66972-C5-3-R,” and by the Swedish Research Council (VR), grant no. 621-2012-5332.

is an expensive structure in area and energy, but it also inhibits scaling of the microarchitecture: the energy cost of CAM structures grows exponentially with size, but without a corresponding contribution to performance [5].

Prior work concentrated on eliminating the searches in the LQ since the culprit for the high cost is the associative nature of the LQ. A non-associative LQ is possible if stores and invalidations do not have to search the LQ to find conflicting loads. The key insight is that the detection of a conflict between a load and a store (of the same core or a remote core) can be *deferred* until the load tries to commit. To test for a conflict that may have occurred between the time the load was issued and the time it commits, the load is *replayed* and the loaded value is *re-checked* (i.e., re-loaded from the L1). This ensures that even when conflicting stores or invalidations are no longer available at the time of commit, the conflict will not go undetected. This approach was first proposed by Gharachorloo et al. [1] but was quickly dismissed as it puts tremendous pressure on the L1. Cain and Lipasti [4], improved the idea by filtering out all the loads that need *no replay* simply because there was no store with unresolved address, nor any invalidation received, between the loads' issue and their commit. The technique was further improved by Roth [6] with more accurate filtering of loads using the concept of Store Vulnerability Window (SVW). Despite the successive improvements, there remains a sizeable fraction of loads that need to be re-executed at commit. This increases L1 energy consumption, creates interference in the cache ports, and ultimately hurts performance.

This paper proposes two solutions to completely eliminate the load queue and its limitations without affecting the L1 at all. The solutions are based on the observation that stores can be safely—correctly— delayed in the store buffer (SB). Previous approaches ignored the existence of a store buffer and instead considered only the LQ/SQ system in relation to the L1 (cache hierarchy). This misses out on a fundamental property of the SB: we can delay stores in the SB until some condition is met, without affecting the memory consistency model (in our case TSO).

The key observations that drive this work are:

- We can validate load speculation (at commit time) in our own core, *without ever leaving the core to go to the L1*, if we delay our own stores in our own SB, *and*
- We can fend-off any external conflicting stores, whose invalidations would otherwise squash our loads, by delaying *external* stores in their *remote* SBs.

The implication of these two observations is that an LQ is no longer required for the correct operation of an out-of-order core supporting TSO. Specifically:

1) Searching the LQ is no longer required for loads issuing under a memory-dependence speculation (i.e., over older stores with unresolved addresses) as we use the concept of on-commit value-check [1], [4]. The advancement over all previous work is that our solution never requires accessing the L1 for the value re-check. An extra benefit of our approach is that it allows *any* state of the art memory dependence predictor [7], [8] to be used in the core, something that was not possible in previous value-based approaches [4].
2) Searching the LQ is no longer required for loads issued under a consistency speculation (i.e., over older non-performed loads in TSO) as we employ the concept of delaying conflicting stores on invalidation [9]. (Similarly for eviction of cache lines speculatively accessed by loads.) The advancement over previous work is a new approach that does not need an LQ to set, maintain, and track "lockdowns" based on completion of loads. Instead, we implement *cacheline* lockdowns. A cacheline lockdown means: no change (e.g., invalidation or eviction) is allowed for the cacheline as it is under a speculative read. As in the work of Ros et al. [9], a locked-down cacheline will not respond to invalidations; in contrast to that work the cacheline cannot even be evicted.

Altogether, our two-pronged approach completely eliminates the searches in the LQ and therefore the LQ itself. Without an LQ, the commit order of loads is simply kept by the reorder buffer (ROB). A load replay is required on commit to re-check the loaded value for memory dependence speculation. But this replay is restricted in-core: it accesses only the store buffer and never goes to the L1. While for memory dependence speculation we essentially substitute an associative search of the LQ with a search of the store buffer, the benefit comes from relieving the L1 from the burden of load-replay (value re-check). The substantial benefit of our approach, however, comes from eliminating all the LQ searches from external invalidations and cache evictions. External searches account for the bulk of the LQ searches both in previous work [4], [6], and in our own results. No load replay is required in these cases, so all eliminated searches are net profit.

Results: Eliminating the LQ results in significant energy savings of 17.6% for SPLASH, 18.7% for PARSEC, and 22.1% for SPEC, for the structures involved, compared to a value-based replay proposal; and of 5.7% for SPLASH, 5.1% for PARSEC, and 8.3% for SPEC compared to a standard architecture featuring an associative LQ. In addition, the elimination of the LQ in a small core where LQ stalls are an issue, improves performance by up to 10% and 5.9% compared to the value-based approach and the associative LQ approach respectively.

II. Background

Conceptually, a load queue can be constructed as an *insulated* structure. Insulated LQs perform all operations in strict program order and thus are low performance. An insulated LQ requires snooping on load issue, but on the other hand, it does not need to be searched.

In this paper, we consider high-performance LQs that need to be searched. These LQs support uniprocessor memory dependence speculation and multiprocessor speculative memory

reordering. In a searchable load queue, the searches are performed in two situations: 1) when a store resolves its address to validate a load's *memory dependence speculation*, and 2) upon an invalidation or a cache eviction, to enforce a memory consistence model. This section gives some background for work relating to these two situations.

A. Memory Dependence Speculation

Instruction Set Architectures promise that instructions appear to be executed atomically and sequentially in program order. Naturally, a load must take its value from the most recent —in program order— store to the same address that precedes the load. In an out-of-order core this translates to every load associatively searching the SQ and the SB for the latest store on the same address if it exists in there. This *store-search* is fundamental for the ISA and cannot be eliminated. Without a match in the SQ/SB the load accesses the L1.

The memory dependence problem appears when the address generation for a store instruction is delayed (perhaps due to a long latency miss) and younger loads are eagerly and speculatively issued assuming that their address does not match the unresolved address of the store. Such loads are called *D-speculative* according to the terminology of Ros et al. [9]. A D-speculative load requires at least two associative searches and one access to the L1: i) a load searches the store queue/store buffer (SQ/SB); ii) speculatively accesses the L1 (assuming no store forwarding from the SQ/SB); iii) at least one store resolving its address accesses the LQ to check if the D-speculative load violated a memory dependence. (This count does not include the searches and accesses that follow if the load is squashed and re-executed.)

Associative searches with Memory Dependence Prediction: Memory dependence prediction (MDP) [7], [8], [10], provides an effective way to address the memory dependence problem, by predicting when loads are likely to depend on older stores and thus avoiding the likely-futile issue of the loads when these older stores have not resolved their address. Memory dependence prediction is only "safe" when it predicts the existence of a dependence, as it conservatively prevents a load from executing with a potentially wrong value. If the prediction is wrong (and there is no dependence) no harm is done other than delaying the execution of the loads. However, when MDP predicts absence of dependence, it is *unsafe* as it allows loads to execute in the presence of stores with unresolved addresses. If the prediction is incorrect, and there is a dependence, then issuing a younger dependent load is incorrect and the load must be squashed (this typically means that all instructions after an incorrect load are squashed). Because the likelihood of predicting absence of a dependence is higher in many workloads there is only modest reduction in the associative searches of the SQ/SB and the LQ. We assume MDP throughout this paper.

B. Eliminating LQ Searches with a Value-Based Approach

A value-based approach [1], [4] eliminates the load-squash searches in the LQ, by delaying the possible squash until a load attempts to commit. At commit, a load rechecks its loaded value by reloading from the L1 in case an intervening store did not change the value. This means that one associative search and up to two access to the L1 are performed per D-speculative load: i) a load searches the store queue/store buffer (SQ/SB); ii) speculatively accesses the L1 (assuming no store forwarding from the SQ/SB); iii) re-accesses the L1 on commit to validate its loaded value. (This count does not include the accesses if the load is squashed and re-executed.)

Compared to the standard way of handling speculative loads described in Section II-A, we simply trade a search of the LQ with an access to the L1. How profitable is this tradeoff depends on the relative costs of searching the LQ versus accessing the L1.

In any case, it is imperative to reduce the replays at commit to a minimum, as the increased pressure on the L1 is detrimental to performance. To avoid replaying all loads at commit time, Cain and Lipasti [4], use a natural filter for this case: loads are only required to replay at commit if a store with an unresolved address exists in the SQ (not committed) at the time of the load's issue. Roth [6] improves this natural filter by taking into account the ordering between the loads and stores (with an elaborate numbering scheme for stores) and enforcing the replay only if an *older* store with an unresolved address is present in the SQ at the time of the load's issue. Despite the successive improvements no proposal managed to bring a value-based approach on par with the base architecture in terms of performance. The extra L1 accesses tax an already strained resource and diminish the possible energy benefits from converting the LQ into an non-associative structure.

C. Enforcing Consistence and Coherence on Speculatively Reordered Loads

Forcing memory operations to execute in program order is too restrictive for performance even for strong consistency models such as TSO. Out-of-order cores allow loads to execute speculatively out-of-order with respect to older non-performed loads. Such reordered loads are called M-speculative [11]. However, consistence can be violated when a conflicting store from another core races with a speculatively reordered load. When this is detected by the invalidation of the conflicting store, the speculative load must be squashed and re-executed in its proper order [12]. In addition, a cache eviction at the time a load is speculatively reordered is also cause for squashing the load. The reason is that without the L1 cache line the invalidation of a conflicting store will not be received. The solution for these cases requires an associative LQ that can be searched for a matching speculative load.

A value-based approach, defers the validation of an M-speculative load for the commit stage, where the load is replayed and re-checks its value [4], [6]. If the value has changed from the speculatively loaded value, the load is squashed. Similarly to the case of the dependence speculation discussed above (Section II-B), a value-based approach obviates the need for a searchable (associative) LQ, but substitutes the associative search of the LQ with an extra L1 access. Not all

M-speculative loads need to be replayed, though. A natural filter for this approach is to replay only the M-speculative loads in the presence of an invalidation or cache eviction.

D. Non-Speculative Load Reordering

Recent work by Ros et al. [9] shows that preserving the appearance of the load order in TSO can be efficiently achieved by the coherence protocol in a non-speculative way. The problem in TSO is that a race between a load and store might happen at a time when the load can be "seen" as reordered. A reordering of a younger performed load can be observed by a conflicting store simply because an older load is not yet performed. The key observation of Ros et al. is that if the conflicting store occurred slightly later, after the older load is performed, the reordering could not have been observed. The proposal of Ros et al. is to simply delay the conflicting store in its store buffer (by withholding its invalidation acknowledgment) until the reordering is resolved (all older loads than the reordered load have been performed). While this approach has the potential to catalyze value-based approaches by significantly reducing squash events, unfortunately it relies on the existence of an LQ to keep track of "locked-down" loads (loads that execute out-of-order with respect to older loads) as well as the invalidations they may receive and withhold until their reordering cannot be observed. In the same work Ros et al. also propose an external structure, called Lockdown Table (LDT) for the purpose of removing loads from the LQ (out of order commit), however, the LDT is nothing more than a external proxy of the LQ for the removed loads. The advancement of our work is to achieve the same goal *without using an LQ*.

III. Delaying Stores in the Store Buffer

In this section, we describe our approach to eliminate the LQ in a processor supporting TSO. We first describe the processor model that we use; subsequently, how LQ searches for the validation of memory dependence speculation are eliminated; and finally, how locking cachelines in the L1 and delaying conflicting stores in their store buffer eliminates the need to search the LQ for invalidations and cacheline evictions.

A. Base Processor Model

As the base processor model we assume a typical out-of-order architecture that includes an instruction queue (IQ), a reorder buffer (ROB), a load queue (LQ), store queue (SQ) and store buffer (SB). During the dispatch stage, instructions dispatched to the IQ are inserted in the ROB and depending on whether they are loads or stores are also inserted in the LQ and the SQ respectively. If any of these structures cannot accommodate an insertion, the dispatch (and possibly the whole front-end) is stalled.

The LQ and SQ replicate instructions that are in the ROB. They are simply helper structures and their function is to hone in various searches to the proper subset of instructions. In other words, one can do without an LQ and an SQ if one is willing to associatively search the entire ROB. But that would be too expensive. Instead, having a separate LQ and SQ means that the ROB does not need to be searched and can be structured simply as a circular memory buffer (SRAM). The relative ordering of the loads in the LQ and the stores in the SQ is defined by their position in the ROB. This is explained in Section III-C.

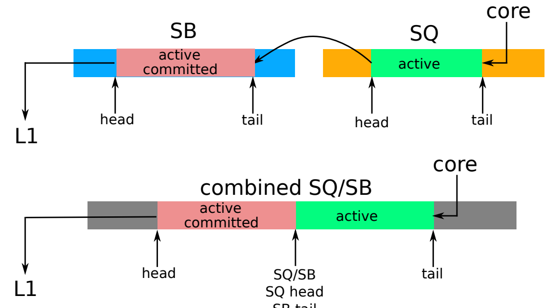


Fig. 1. Combined SQ/SB

The SB (Fig. 1, upper left) is the structure that holds *committed* stores. When a store reaches the head of the ROB (and the head of the SQ) and commits, it transitions from the head of the SQ to the tail of the SB (Fig. 1, upper diagram). The store is made visible in the memory system when it reaches the head of the SB and is written in the L1. If the corresponding cacheline is present in the L1 and has write permissions, the store is completed and removed from the SB. A miss, or a *write miss* (no write permissions) in the L1 initiates a coherence action. To overlap the latency of a possible coherence action with the queuing time of a store in the SB (the time to reach the head of SB), a prefetch for permissions is sent as soon as a store enters the SB. Actual processors issue such prefetches at this time [13].

While we are discussing the SQ and the SB as two separate structures, they can be one and the same, as found in some implementations [13]. A combined SQ/SB is a single physical structure that is logically divided into an SQ part and an SB part (Fig. 1, lower diagram). The whole structure is a single FIFO CAM buffer. Stores enter in the tail pointer at dispatch and are removed from the head pointer when they are written in the cache. The logical distinction between the SQ and the SB part is kept with an additional pointer that separates the non-committed from the committed stores.

The benefit of the unified (combined) SQ/SB structure is a better utilization of the available buffer space [14]. The downside is that it is more difficult to perform targeted searches in just one part (e.g., the SQ or the SB). To perform such targeted searches we would need to modify the CAM circuitry to first select based on a single distinguishing bit which part of the structure will actually perform the rest of the comparisons [15]–[17]. In this paper, we treat the SQ and the SB as a combined structure.

In this base architecture the LQ is a necessity as it is searched every time a store resolves its address, or an external invalidation is received, or a cache eviction occurs. In the rest of this section we describe our proposal without an LQ.

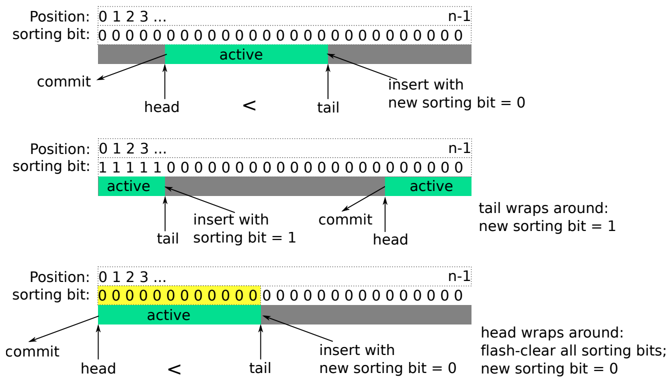


Fig. 2. Instruction order: ROB position encoding and sorting bit

B. Base Processor Model with Replay on Commit

The base processor model with replay on commit is modelled after the Cain and Lipasti work [4]. In this model speculative loads re-check their value in two extra pipeline stages before commit. In the first of these stages the L1 is accessed (Cain and Lipasti [4] assume that the replay L1 access can be performed faster than the initial L1 access) and in the second stage the speculative value is compared against the re-check value.

C. Instruction Order

An important requirement for our approach, that will become evident in subsequent sections, is the ability to easily determine instruction order, for example between loads and stores. This could be accomplished by giving each dynamic instruction, at the time of dispatch, an ascending sequence number (e.g., as in [6]), but there are two problems: i) for large sequence numbers, storage and energy costs can add up, and ii) an implementation with a limited-width counter for the sequence numbers incurs frequent wraparounds.

We only have to compare instructions that are in the processor's instruction window at the same time —in other words, instructions that are never more than n places apart, where n is the size of the ROB. This allows us to use an efficient encoding inspired by the proposed encoding of Buyuktosunoglu et al., for ordering instructions in a non-compacting instruction queue [18]. In the Buyuktosunoglu encoding, an instruction's order is simply its ROB position augmented by an additional high-order bit, called the *sorting bit*. The sorting bit is needed because the ROB is a *circular* FIFO buffer and both the head and the tail pointer can wrap around the end of the ROB to its beginning. Thus, we are faced with the situation where newly inserted *younger* instructions occupy lower positions in the ROB than older instructions. The sorting bit correctly encodes this situation in the instruction's order.

In our case, the mechanics of the sorting bit are as shown in Fig. 2: In the beginning, where the head and the tail pointer are ordered (head ROB position $<$ tail ROB position) each dispatched instruction receives a sorting bit of 0 and is inserted at the tail (Fig. 2, upper diagram). When the tail pointer wraps around the end of the ROB (head ROB position $>$ tail ROB position), the sorting bit changes to 1 (Fig. 2, middle diagram). This gives the newly inserted instructions in the lower ROB positions, a *higher order* than the ones starting at the head pointer. When the head pointer wraps around the end of the ROB, all the sorting bits of all the entries in the ROB are flash-cleared to 0 (Fig. 2, lower diagram).

An instruction's order of $log_2(n)+1$ bits (i.e., ROB position + sorting bit) is kept in the instruction's ROB entry, and is copied where needed.[1] The sorting bit is flash-cleared, when needed, in all structures where the ROB order of an item has been copied. In the rest of this paper, we will simply compare instructions (younger versus older), implying that a comparison of their ROB-encoded order is taking place.

D. Validating Memory Dependence Speculation

When a load issues over an older store with an unresolved address we are speculating that there is no memory dependence between the two. This speculation can be validated in one of two ways:

- have the store check the LQ for a matching younger load,
- or have the load re-check its loaded value at the time of commit.

The first case necessitates the associatively-searched LQ of the base architecture as it is the store's responsibility to find any younger loads that may have executed speculatively and need to be squashed. The search occurs when the store resolves its address.

In the second case, it is the load's responsibility to check the speculation. Speculation is initiated at the load's issue time, when by accessing the SQ we determine the existence of prior stores with unresolved addresses [4], [19]. However, at the load's commit time, the store that caused the speculation has already committed (as it is older than the load which is about to commit) and may have already gone to the memory system. This means that the load may have to re-check its value by looking in the SB *and* the L1 (failing to find a match in the SB).

The key idea to eliminate the load-replay L1 accesses is to delay the appropriate stores in the SB until the loads that were issued under a memory dependence speculation have a chance to re-check their value solely in the SB.

E. Sentinels

To simplify the presentation, we first explain the main idea from the perspective of a single speculative load that issues over a single store with an unresolved address (Fig. 3).

On a load issue, we search the SQ/SB for the most recent store on the same address, or an unresolved address. Recall that we assume a unified SQ/SB structure, but our approach can be easily adapted for separate structures. The SB part of the SQ/SB cannot have stores with unknown addresses, but the SQ part can.

[1] Committed stores in the SB are outside the instruction window and do not make use of this encoding.

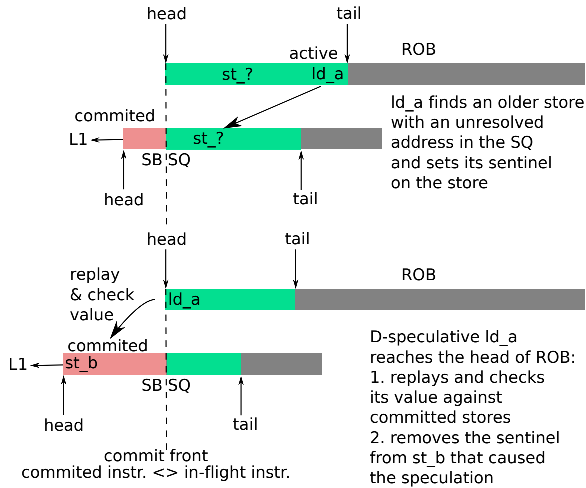


Fig. 3. A ld sets a sentinel and replays on commit as D-speculative (Without loss of generality, for all remaining figures we will draw the head pointer of the ROB and the SQ/SB on the left edge of the corresponding physical structure)

Not all loads need to re-check their value. Only if a load matches an unresolved-address store in the SQ (and without an intervening store on the same address), becomes *dependence-speculative* (*D-speculative*) and cannot commit unless its speculation is validated with a replay.

In such a case, we ensure that the load's speculation will be validated *while the store is still in the SB part* as follows:

- We mark the unresolved-address store with a *sentinel* (Fig. 3, upper diagram). The sentinel is the ROB order of the load as discussed in Section III-C.
- We mark the load in the ROB with the position of the store in the combined SQ/SB. This position does not change when the store commits.
- When the store resolves its target address, it does not search the LQ for speculative loads —it is the load that is responsible to re-check its value.
- The store can commit (pass from the SQ part to the SB part) without any further action.
- However, a committed store *carrying a sentinel* that reaches the head of the SB cannot be written in the L1 until the load that set the sentinel re-checks its value. A blocked store at head of the SB blocks all younger stores. This cannot result in a deadlock as it is explained in Section III-G.
- On commit, the D-speculative load re-checks its value by searching the SB (Fig. 3, lower diagram). It does not need to check the SQ, since any older store has already committed by the time of the load's commit (see the dividing line "*commit front*" in Fig. 3).[2] A D-speculative load searches the SB part for a matching address.

[2]As we mentioned previously, it is more difficult to check only the SB part of a combined SQ/SB than it is to check a separate SB, but with the appropriate modification of the CAM logic [15]–[17] it is possible to do as an optimization.

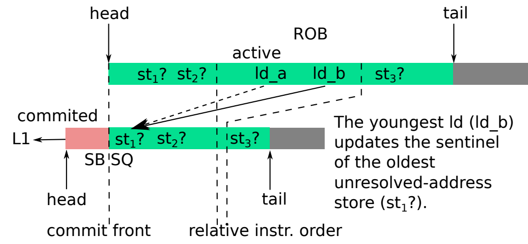


Fig. 4. Setting and updating sentinels

- When the D-speculative load re-checks its value it removes its sentinel from the store and the store is allowed to exit the SB and write the L1. Removing the sentinel is a direct access to the SQ/SB as the position of the store with the sentinel is known to the load.

To summarize: a store resolves its address while in the SQ; if it is matched by a load before that point, the store is marked with a sentinel (the load's ROB order) and cannot leave the SB until the same load reaches the commit stage and re-checks its value. At that point the load will find that the store's address is either: *the same* (a dependence violation) and the load (and subsequent instructions) must be squashed and re-executed;[3] or *different* (no dependence), in which case the load commits.

F. Setting and Updating Sentinels

The discussion thus far is from the perspective of a single D-speculative load and a single unresolved-address store. However, the situation in out-of-order execution can be significantly more complex with multiple D-speculative loads and multiple unresolved-address stores at the same time. In particular, we discussed how a load sets a sentinel on a store but what happens when multiple loads need to set sentinels on the same store? Similarly, what happens when a load is under the shadow of multiple stores with unresolved addresses? Despite the chaos of out-of-order execution, our approach is based on a very simple principle: *A sentinel is always set between the youngest issuing load and the oldest unresolved-address store.*

From a store's perspective:

- A store in the SQ with an unresolved address can be marked with a sentinel by any newly issued load. The only condition when a store's sentinel is replaced is that the new sentinel must be *younger* than the old (e.g., `ld_b` in Fig. 4).
- Once a store resolves its address, its sentinel cannot be updated any longer, simply because the store cannot be matched as an unresolved-address store. The store will commit (pass from the SQ to the SB) with its current sentinel and it must be matched while in the SB by the corresponding load.

From a load's perspective: At issue, the load searches the SQ/SB to match stores (older than the load) with the same

[3]A trivial opimization here is to check if the store is *silent*, i.e., if it writes the same value as the one loaded speculatively. In this case no squash is needed.

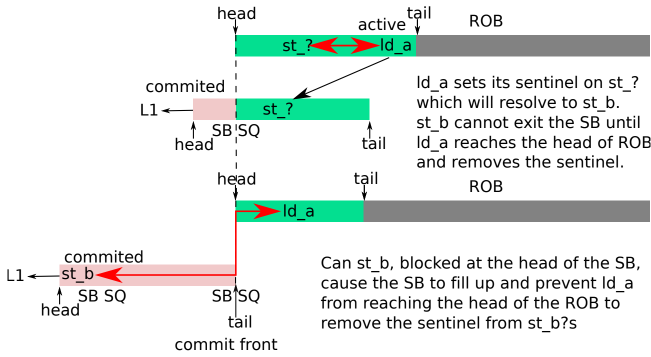


Fig. 5. No deadlock is possible by blocking the head of the SB

address or unresolved addresses (which can be thought as wildcards). The following cases exist:

- There are one or more stores with a matching address: the load must take its value from the *youngest* of these stores only if there is no other store with an unknown address, even younger than that.
- There are one or more stores with an unresolved address, younger than any store with a matching address: the load must set its sentinel on the *oldest* of these unresolved-address stores (e.g., $\texttt{st}_1\texttt{?}$ in Fig. 4) and save the store's position in its ROB entry for the replay.

Because we assume that the SQ/SB is a non-compacting FIFO CAM structure, the operations described above require oldest-first selection logic (to select the oldest among unresolved-address matches) and its reverse, youngest-first selection logic (to select the youngest among address matches). In both cases, the selection is based on the ROB order of the stores (Section III-C). Such age-based selection logic is described in detail by Buyuktosunoglu et al. [18]. In case both selection circuits produce a result, an additional step is required for the final selection. This step is typically just a simple comparison or the ROB order of the two results (the oldest unresolved-address match against the youngest address match) and in rare cases an additional selection of an unresolved-address match may be needed for setting the sentinel on the correct store. This additional step, however, is off of the critical path of the load.

G. Absence of Deadlock

Blocking a store at the head of the SB raises concerns for deadlock. In particular, the situation that can lead to a deadlock is when a store blocking the head of the SB causes the SB to fill up (consume all the space of the combined SQ/SB), leaving no space in the SQ to accept new stores. This situation is depicted in Fig. 5. The question is: can this prevent the load that set the sentinel from reaching the head of the ROB, replaying, and removing the sentinel from the blocked store?

Consider, first, that the SQ contains exactly the same stores as the ROB. Any store that is inserted in the ROB is also inserted in the SQ at the dispatch stage. If the SQ cannot accept any more stores, the front-end stops inserting instructions in the ROB (stalls). For deadlock to occur the *distance* of `st_b` and `ld_a` , when counted in the number of stores (depicted by a red double arrow in Fig. 5), must be *larger than the maximum size the SQ can take (i.e., larger than the size of the combined SQ/SB)*.

However, the distance between the load and the store is fixed while they are both in the ROB, at the moment `st_?` resolves its address, becomes `st_b`, and prevents any further update of its sentinel by any younger load. The store resolves its address while still in the SQ; it could not have been committed with an unknown address. Thus, deadlock is impossible as it implies that at some point *more* stores were inserted in the ROB than what the maximum size of the SQ can fit.

H. Fending-Off Conflicting Stores

The non-speculative load reordering approach of Ros et al. [9] delays conflicting stores in their SB so that a reordering cannot be observed and violate a consistency model (such as TSO). Delaying a conflicting store in its store buffer is achieved by withholding the acknowledgement to its invalidation, thereby preventing the store from obtaining write permission as it tries to exit the store buffer and write to its L1. At the coherence level, a new directory state called WriterBlock and the use of non-cacheable tear-off data guarantee absence of deadlock and livelock [9].

The important feature of the approach, however, relevant to our work, is that speculative loads do not need to be squashed by invalidations or cache evictions.

In the value-based approach of Cain and Lipasti [4] invalidations and cache evictions are responsible for causing the replay (in the L1) of 20-40% of all loads in their workloads, while the corresponding percentage for uniprocessor dependence speculation is only 2-15%. Even with sophisticated (but complex and costly) re-execution filtering techniques introduced by Roth [6], more than 15% of the loads have to be replayed in the L1. A technique that avoids the squashing of loads on invalidations and evictions is therefore invaluable as it obviates replay for these cases.

The problem is that the Ros et al. approach is intimately connected to the LQ, as we mention in Section II-D. More specifically, the LQ is used to: i) determine when a load is reordered with respect to older loads; ii) collect any invalidation that matches one or more loads; iii) determine when a load becomes ordered with respect to older loads and release the invalidation acknowledgement to unblock the corresponding conflicting store. In our approach, there is no LQ to provide this functionality. Instead, lockdowns are implemented directly in the L1 which is also accessed by invalidations. We determine if a load issues over older non-performed loads by selectively ORing status bits from the load in question to the head of the ROB. A load that speculatively issues over older non-performed loads, accesses the L1 and sets its sentinel in the accessed cacheline. When the load receives the data from the L1, it records in its ROB entry the location of the cacheline in the cache (way and index). The cacheline goes in lockdown mode which means that it is under

a speculative read and cannot be invalidated nor evicted. The sentinel can be updated by younger speculative loads, up until the time the cacheline is matched by an invalidation. At this point the sentinel cannot be updated any longer. This prevents a livelock from an endless update of the sentinel. A cacheline in lockdown that receives an invalidation, withholds the Ack to the invalidation until the time that the speculative load that *last* set the sentinel reaches the head of the ROB and commits. At that point the load removes the sentinel form the cacheline, with a direct access, using the location information stored in its ROB entry.

Updating the sentinel is no different than updating state bits of the tag (for example, coherence or replacement bits) which is a frequent operation. Thus, when updating tag state, we can set a new sentinel or replace an older sentinel. Upon receiving an invalidation (or on an eviction), the L1 cache controller checks for the existence of a sentinel, alongside with the checking of other state bits (coherence/ replacement bits). When a snoop hits a valid sentinel, the specific sentinel is simply not allowed to change any more. This takes one extra state bit to enforce. Snoops never pass the L1 to go into the core.

When a sentinel is removed from a cacheline, any invalidation acknowledgement that was withheld is released. Acknowledgments are queued in the L1 coherence controller and simply wait for a signal form the corresponding cacheline (that enqueued them) to return.

As with the proposal by Ros et al., the coherence layer ensures that loads are not blocked anywhere in the memory system, using WriterBlock states in the directory and, if required, non-cacheable tear-off data. The latter is particularly evident in our solution for evictions. Recall that evictions are not allowed for cachelines in lockdown. In such cases, a conflict with a lockdown cacheline, turns into a non-cacheable access.

Our approach achieves the same functionality with non-speculative reordering of loads but without an LQ. In this respect, it is simpler that the Ros et al. proposal. No replay is required for M-speculative loads that reach the commit stage as it is guaranteed that the cacheline has not changed in any way. Instead, only the removal of the sentinel is required, which takes place in the same pipeline stages used for the replay of D-speculative loads. The end result is a substantial reduction of replays compared to the previous value-based approaches.

IV. Sentinel Removal on Squash

Thus far, we discussed the removal of sentinels (from the SB or the L1) when the corresponding loads (D-speculative or M-Speculative respectively) reach the head of the ROB. Note that *mispeculated* D-speculative loads always flow to the head of the ROB to be re-checked and only then can be squashed, while M-speculative loads are never squashed by invalidations or evictions. However, an important question is how do loads remove their sentinels if they are squashed by any other reason: control mispeculation, exceptions, or mispeculation by prior instructions.

The sentinel removal technique depends on how the processor squashes and discards mispeculated instructions: lazily or eagerly.

A. Lazy Squash (a.k.a. "Bogus Retirement")

In lazy squash, squashed instructions are marked *in place* in the ROB and are discarded when they reach the head of the ROB. Thus, each and every squashed load removes any sentinel it may have set (in the SB or the L1) when it reaches the commit stage and is discarded.

B. Eager Squash

Our approach is equally applicable in architectures with *eager squash* where instructions are squashed in bulk, their ROB entries are immediately reclaimed and allocated to newly issued instructions. However, in this case, the removal of sentinels is more involved.

Assume that an instruction (e.g., speculated branch, D-speculative load, etc.),[4] henceforth called *point-of-squash* causes a squash of all subsequent instructions. All the sentinels *after* the point-of-squash (i.e., younger) must be canceled in the SB and in the L1. However, this cannot happen before the point-of-squash reaches the head of the ROB so we can guarantee that all the instructions *before* the point-of-squash (i.e., older) had their chance to re-check on commit:

- D-speculative loads flow to the head of the ROB to be re-checked and only then can become the point-of-squash. In this case, there are no sentinels left before the point-of-squash and, therefore, all remaining sentinels are *flash-cleared* both in the SB and in the L1.[5]
- In all other cases, we wait for the point-of-squash to reach the head of the ROB, guaranteeing that any older instruction, before the point-of-squash, is able to re-check (if needed) on commit. At that point, we do a flash-clear or all remaining sentinels in the SB and in the L1 (and release any blocked invalidation acknowledgments).

While waiting for the point-of-squash to reach the head of the ROB we *choose*, for simplicity, *not to set any new sentinels*.[6] While this implies that we may have to delay issuing speculative loads, this practically does not happen. Our data shows that the time it takes for the point-of-squash to reach the head of ROB is almost always *shorter* than the time a new sentinel would be needed after the squash. In the vast majority of our benchmarks, almost 100% of the mispredicted branches reach the head of ROB before a new sentinel needs to be set in the correct path. We only found few benchmarks with a notable percentage of the mispredicted branches that delay sentinels, on average for very few (less than 3) cycles, see Table I.

[4]Exceptions are handled as mispeculated branches

[5]Only one bit per SB-entry or L1 cacheline needs to be cleared which is feasible.

[6]New sentinels *can* be set but the overlapping numbering with the old set of squashed sentinels requires complex handling.

TABLE I
MISSPREDICTED BRANCHES THAT CAUSE STALLS OF SPECULATIVE LOADS

Benchmark	Misspredicted branches	Stall speculative loads
lu-ncp	0.59%	11.4%
dedup	1.4%	4.7%
lu-cp	0.59%	1.4%
streamcluster	0.46%	0.9%
canneal	7.8%	0.3%
all other		$< 0.1\%$

C. Benefits and Costs

To summarize, the benefits of our approach compared to a standard architecture with an LQ are:

- Area and complexity: We do not have an LQ.
- D-speculative loads: We perform a replay only in the SB —not in the SQ— instead of a search in the LQ.
- M-speculative loads: We eliminate all LQ searches, replacing each with a direct access to the cache to remove the sentinel.

Compared to the Cain/Lipasti [4] and Roth [6] replay:

- We have the same filtering of loads to replay for dependence speculation as with the Store Vulnerability Window [6].
- We have no L1 accesses for the replay of D-speculative loads (we search the SB instead of the L1).
- We have no L1 replay accesses for M-speculative loads that are matched by an invalidation but simply a removal of the sentinel with a direct access.
- We are free to use memory dependence prediction that correlates loads to stores [7], [8] as we know the conflicting store in the SB. In contrast, in previous value-based approaches the identity of the store is lost when it writes the L1 and a conflict detected by the re-checked L1 value provides no useful information about the store (see the discussion in [4]).

On the other hand, we incur a number of costs:

- We store ROB order, SQ/SB position of stores and L1 position of cachelines, in ROB entries to enable the direct accesses.
- We store sentinels in the SQ/SB and in L1 cachelines.
- Selection logic in SQ/SB searches for both oldest and youngest entries.
- Finally, we need an extra search port in SQ/SB to handle the increased contention due to replay.

V. EVALUATION

Our simulation infrastructure is based on the cycle-accurate GEMS simulator [20] for multicore systems, which offers a timing model of the memory hierarchy, the cache coherence protocol, and the interconnect (GARNET [21]). A detailed x86-like in-house out-of-order processor model driven by a Sniper [22] front-end has been incorporated into GEMS. The processor model implements a TaglessCHT memory dependence predictor [10] and a Tournament branch predictor [23].

TABLE II
SYSTEM CONFIGURATION

Processor class: Silvermont (Nehalem / Haswell)	
Issue / Commit width	4 instructions
Instruction queue (IQ)	16 (32 / 60) entries
Reorder buffer (ROB)	32 (128 / 192) entries (Lazy squash)
Load queue (LQ)	10 (48 / 72) entries
Store queue/buffer (SQ/SB)	16 (36 / 42) entries
Memory	
Private L1 I&D caches	32KB, 8 ways, 1 (addr calc) + 3 hit cycles, pipelined, 64 MSHRs, next-line prefetcher
Private L2 cache	128KB, 8 ways, 12 hit cycles
Shared L3 cache	1MB per bank, 8 ways, 35 hit cycles
Directory (8 banks)	512 sets, 8 ways (200% coverage)
Memory access time	160 cycles
Network	
Topology	Fully connected
Data / Control msg size	5 / 1 flits
Switch-to-switch time	6 cycles

The TaglessCHT memory dependence predictor does not rely on the identity of the stores and thus is appropriate for all techniques we evaluate, including the value-based techniques that replay in the L1 (see Section IV-C). Our L1 cache model implements fully pipelined read and write ports and a next-line prefetching [13]. We simulate a multicore processor consisting of 8 out-of-order cores. The architectural details of the simulated system, modeling an Intel Silvermont processor, are displayed in Table II. We also modelled Nehalem and Haswell-class processors but since the LQ is not a bottleneck in these larger cores we only see minor performance improvements. The energy benefits over the baseline, however, are comparable to those of Silvermont.

We run both parallel and sequential applications. The parallel applications are from the SPLASH-3 [24] and PARSEC 3.0 [25] 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 are presented for their parallel region. The sequential applications are from SPEC CPU2006 benchmark suite with the ref input set. Results correspond to the most representative region of 1 billion instructions chosen using the SimPoint methodology [26].

We model four different techniques to guarantee memory dependence and memory ordering. *LQ* is the approach implemented in most commodity processors. It employs a LQ which is searched every time a store resolves its target address or an L1 cache invalidation or eviction happens. *Replay* is the alternative proposed by Cain and Lipasti [4] based on an in-order load replay and value comparison before retirement. Since this alternative does not require searching the LQ, we have optimized the design by completely removing the LQ. In addition, we model replay filter optimizations, and we have adapted the proposal for a TSO consistency model, where the SB also needs to be accessed on load replays. The SB and L1

TABLE III
ENERGY CONSUMPTION PER ACCESS

LQ: 1 read port, 1 write port, 2 search ports	
Search (nJ)	0.000665415
Read (nJ)	0.000501724
Write (nJ)	0.000541147
SQ/SB: 1 read port, 1 write port, 2 search ports	
Search (nJ)	0.000856529
Read (nJ)	0.000541555
Write (nJ)	0.000810883
SQ/SB (replay): 1 read port, 1 write port, 3 search ports	
Search (nJ)	0.000920791
Read (nJ)	0.000627735
Write (nJ)	0.000930685
L1 cache: 2 read ports, 1 write port	
Tag access (nJ)	0.00123128
Read (nJ)	0.0133430
Write (nJ)	0.0139019
L1 cache (replay): 2 read ports, 1 read/write port	
Tag access (nJ)	0.00135552
Read (nJ)	0.0158799
Write (nJ)	0.0161259

cache ports have been adapted to efficiently support replay. In particular, the SB implements one extra search port and the L1 cache replaces the write port with a read/write port. *NoLQCommit* is our approach that also replaces LQ searches with a replay but does not necessitate an L1 replay since it guarantees that no local or remote writes take place. It still requires a replay on the SB on commit for D-speculative loads. Finally, *NoLQEager* is an idealized version of the *NoLQCommit*, optimized by not waiting until the in-order commit stage to replay in the SB. Loads replay when they stop being speculative and there is a free SB search port. No ports are added to the SB and the time stores are delayed is reduced, but on the other hand the mechanism for detecting when loads stop being speculative is not modelled.

The energy consumption for the LQ, SQ/SB, and L1 cache required for each of the previous techniques has been modelled with CACTI-P [27] for a 22nm process technology. Table III shows the number and type of ports and their dynamic energy consumption. "SQ/SB (replay)" applies to *Replay* and *NoLQCommit*; "L1 cache (replay)" is only required for *Replay*.

A. Results

The main benefit of *NoLQ* over the traditional *LQ* implementation is that the LQ bottleneck is removed, and therefore, stalls due to LQ capacity are eliminated. The main benefit over the *Replay* is that it completely removes the expensive L1 cache replays, making for the first time a replay policy efficient and feasible.

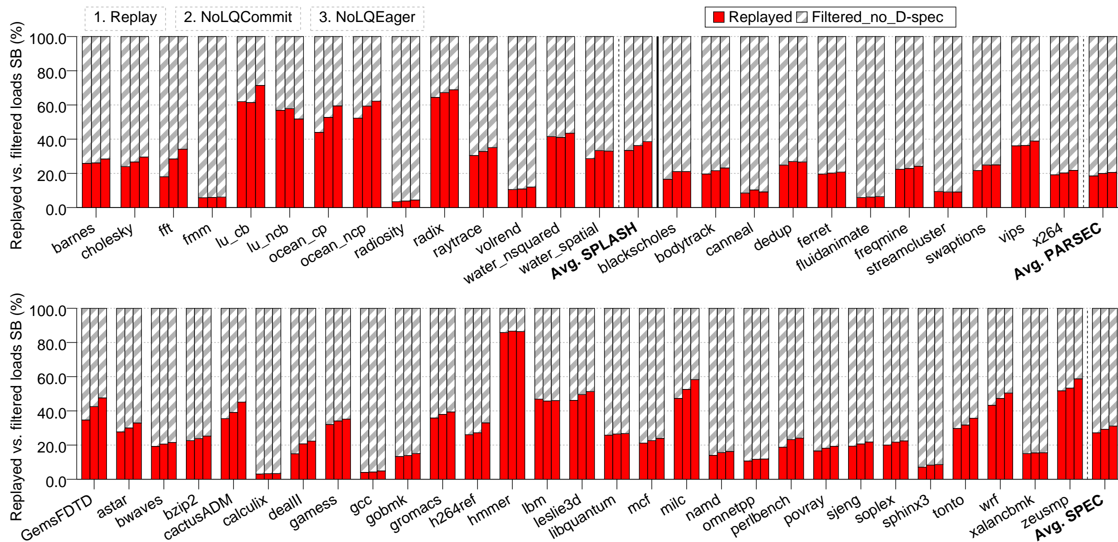
Load replays on the SB. In replay mechanisms, loads executed when previous stores have unresolved addresses (D-speculative) have to perform a replay on the SB to check that the loaded value still matches the value of the previous store (if any). Fig. 6, shows the percentage of loads that replay in the SB and that are filtered since they are not D-speculatively executed. The percentage of replayed loads on the SB is 33.4%–38.6% for SPLASH-3, 18.5%–20.6% for PARSEC and 27.2%–31.1% for SPEC. The percentage of replays increase slightly for the NoLQ schemes since the application runs faster and more loads are executed D-speculatively.

Load replays on the L1. Replays in the L1 only happen in the state-of-the-art replay mechanism, for loads that are performed before older loads (M-speculative) and when the invalidation/eviction filters do not prevent the replay [4]. Fig. 7, shows the percentage of loads that replay in the L1 and loads that are filtered. *Replay* manages to filter most of the replays, but still the percentage of blocks replayed in L1 is 11.2% for SPLASH-3, 7.6% for PARSEC, and 15.2% for SPEC. Even with a low percentage of L1 replays there are three main problems: (i) the L1 cache requires a read/write port, therefore increasing energy consumption (Table III); (ii) replays share the read/write port with writes so extra L1 contention is introduced; and (iii) L1 latency can be long (3 cycles for hits, but for shared blocks replay misses can increase). *NoLQ* filters all L1 replays since it ensures that the data has not been updated since the load was performed.

Processor stalls. Fig. 8 accounts for the percentage of cycles that the processor cannot make progress due to a full ROB, a full SQ-SB, or a full LQ. Processor stalls due to a full LQ only happen in *LQ* (12.4% for SPLASH, 11.2% for PARSEC, and 6.0% for SPEC). *Replay* and *NoLQ* do not incur LQ stalls. In some cases, the lack of LQ stalls translates into fewer overall stalls (as in barnes, water-spatial and swaptions). In other cases the bottleneck moves to the ROB or the SQ/SB (as in ftt, radix, canneal, and lbm). Note that applying out-of-order commit [9] will be even more beneficial for *NoLQ* techniques than for current *LQ* implementations since in *NoLQ* the ROB is by far the predominant bottleneck. We leave this evaluation for future work.

On the other hand, *NoLQ* sets a sentinel in the SQ/SB, preventing stores from performing. This potentially puts more pressure on the SQ/SB. WritersBlock coherence can also add more pressure to the SQ/SB, but it has previously shown to be negligible [9]. As observed in figure 8, the stall percentage due to full SQ/SB does not increase significantly from *LQ* (4.0% for SPLASH, 2.5% for PARSEC, and 0.7% for SPEC) to *NoLQ* (4.4% for SPLASH, 2.8% for PARSEC, and 0.8% for SPEC). Although in *NoLQCommit* the sentinel is released later than in *NoLQEager*, the pressure on the SQ/SB is not affected. Overall, processor stalls are reduced from *LQ* to *NoLQ* by 7.3% for SPLASH, 10.2% for PARSEC, and 9.9% for SPEC.

Execution time. The reduction in the percentage processor stalls translates into lower applications' execution time, as Fig. 9 shows. The extra L1 replays of *Replay* cause however a performance degradation (-8.0% for SPLASH, -3.9% for PARSEC, and -3.5% for SPEC), despite the advantages of not using the LQ. In contrast *NoLQEager* improves execution time compared to the state-of-the-art replay technique (7.8% for SPLASH, 8.3% for PARSEC, and 10.0% for SPEC) and to the LQ technique (3.6% for SPLASH, 4.1% for PARSEC, and 5.9% for SPEC). *NoLQCommit* has an execution time close to *NoLQEager* as the longer blocking time does not considerably affect execution time.



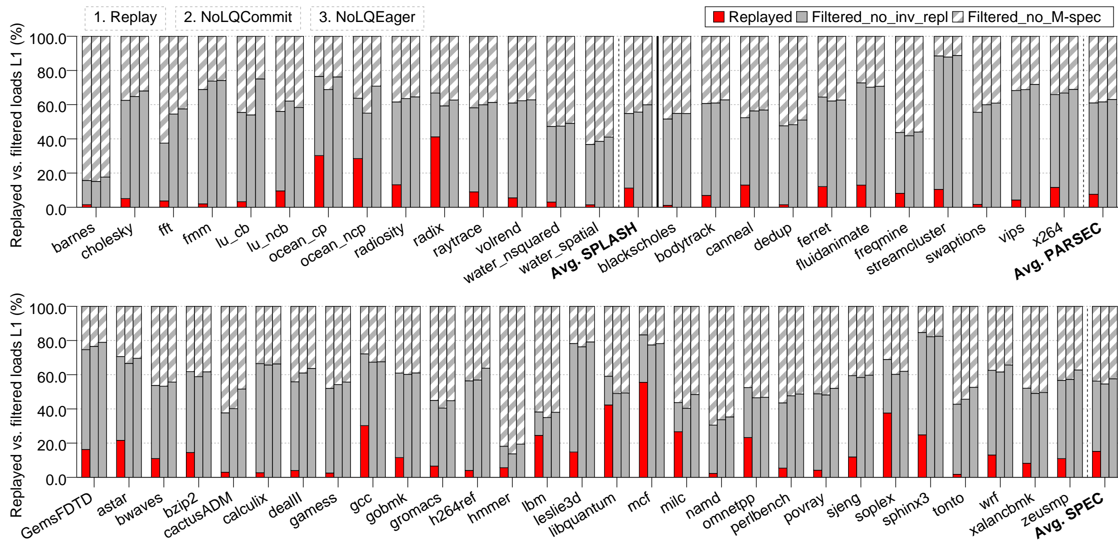


Fig. 6. Load replays in the SB

Fig. 7. Load replays in the L1

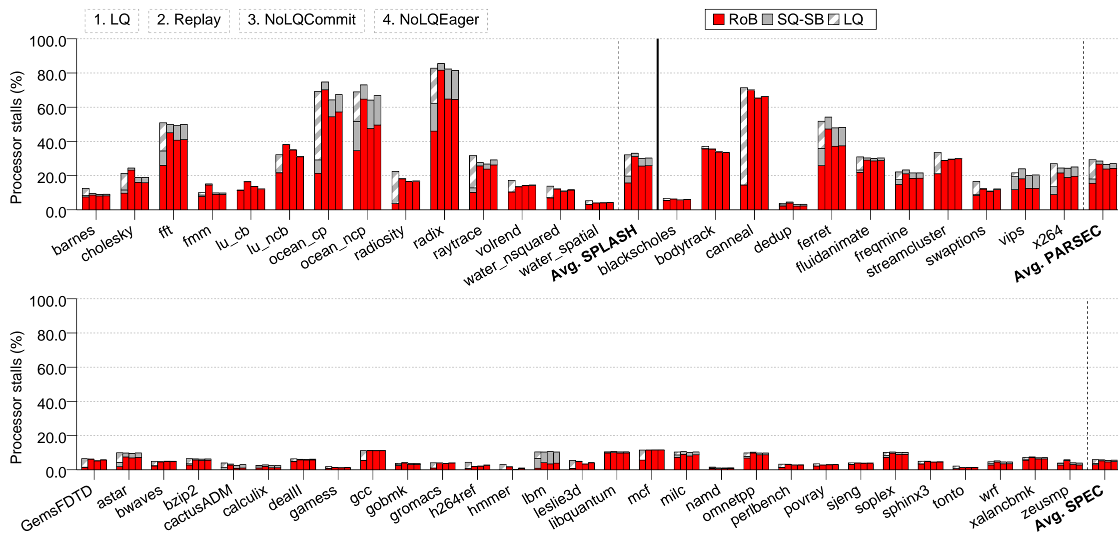


Fig. 8. Processor stalls

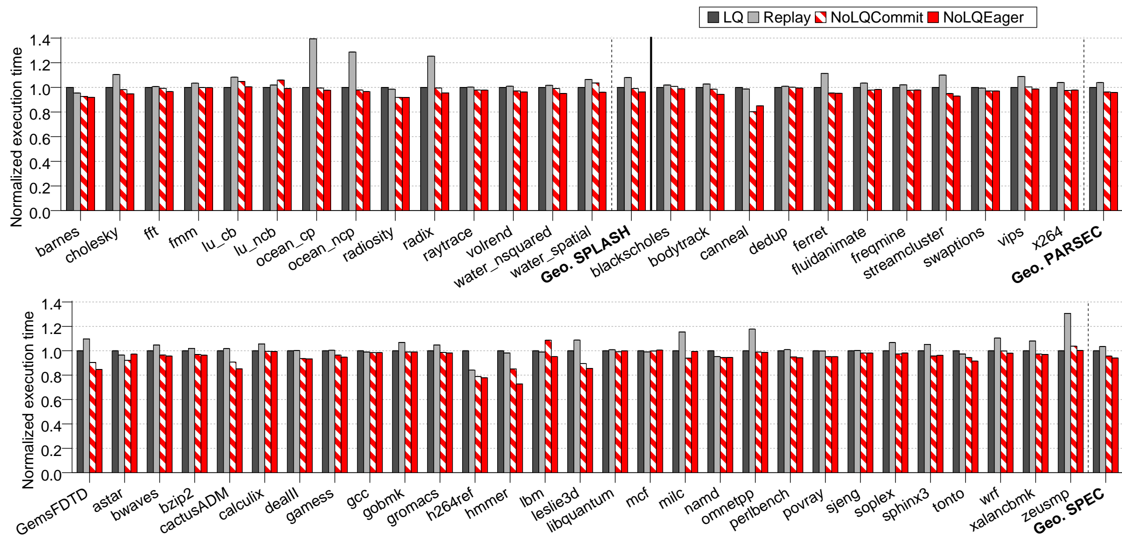


Fig. 9. Normalized execution time

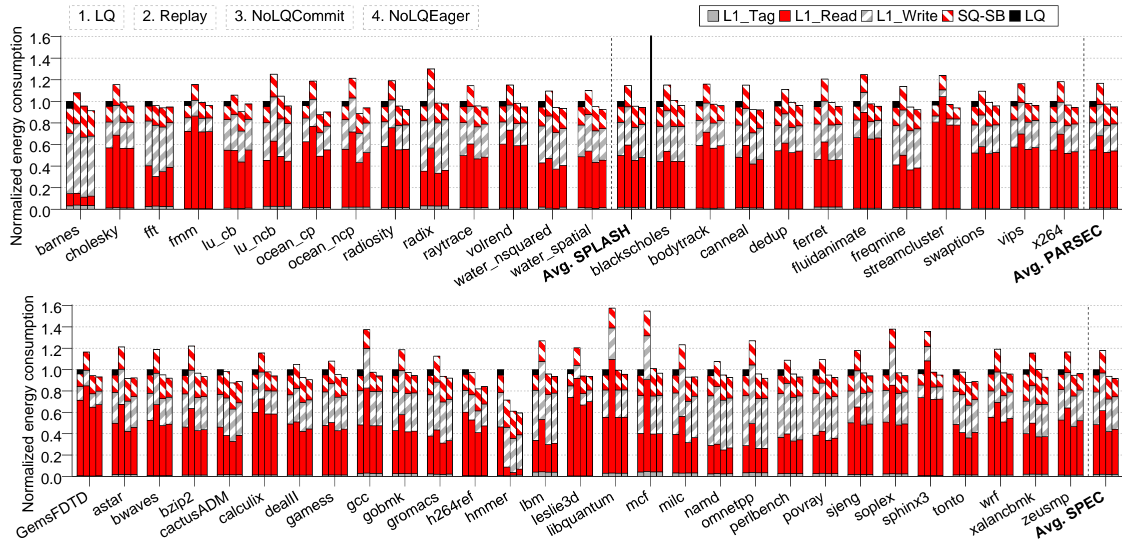


Fig. 10. Normalized energy consumption

Energy consumption. Fig. 10 shows the dynamic energy consumption of the L1 cache, the SQ/SB, and the LQ. L1 access energy is split in L1_Tag (accesses that only check the tag, that is, cache misses or prefetches), L1_Read and L1_Write (hits). Both *Replay* and *NoLQ* save the LQ energy as they do not need such a queue. On the other hand, L1 reads increase for *Replay*. There are some applications where the replay technique increases considerably the L1_Read energy despite a modest number of replays. The reason is that these applications are memory bound and a large fraction of the accesses miss in L1 cache. The replay is frequently a hit (mostly in sequential applications) and that is why the L1_Read increases more than expected. Finally, SQ/SB energy increases in replay techniques, but is not significant since only the committed part of the SQ/SB (the SB) needs to be seached. Overall, *NoLQEager* improves energy consumption of the L1, LQ and SQ/SB both compared to the state-of-the-art replay technique (17.6% for SPLASH, 18.7% for PARSEC, and 22.1% for SPEC) and compared to the LQ technique (5.7% for SPLASH, 5.1% for PARSEC, and 8.3% for SPEC). For Nehalem: 11.3%, 11.3%, 11.8%; Haswell: 13.7%, 15.2%, 15.2% for SPLASH, PARSEC, and SPEC respectively.

VI. Conclusion

The main contribution of this work is to show that the LQ is superfluous for out-of-order execution even under a strong memory consistency model such as TSO. The key concept is to delay stores in the store buffer until we can ensure: i) proper uniprocessor store-to-load forwarding and ii) proper multiprocessor memory ordering. For the former, we employ a novel value-based approach that does not burden the L1 for load replay (value re-check) but instead is confined entirely in the store buffer. For the latter, we employ the concept of delaying remote conflicting stores in their store buffer at the coherence layer but we achieve this in a novel way that does not rely on an LQ.

References

[1] 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.

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

[3] G. B. Bell and M. H. Lipasti, "Deconstructing commit," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2004, pp. 68–77.

[4] H. W. Cain and M. H. Lipasti, "Memory ordering: A value-based approach," in *31st Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2004, pp. 90–101.

[5] V. Zyuban and P. Kogge, "Optimization of high-performance superscalar architectures for energy efficiency," in *2000 Int'l Symp. on Low Power Electronics and Design (ISLPED)*, Jul. 2000, pp. 84–89.

[6] 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.

[7] A. Moshovos and G. S. Sohi, "Streamlining inter-operation memory communication via data dependence prediction," in *30th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1997, pp. 235–245.

[8] 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.

[9] 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.

[10] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan, "Speculation techniques for improving load related instruction scheduling," in *26th Int'l Symp. on Computer Architecture (ISCA)*, May 1999, pp. 42–53.

[11] 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.

[12] 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.

[13] Intel, "Intel® 64 and ia-32 architectures optimization reference manual," www.intel.com, Jun. 2016.

[14] A. Ros and S. Kaxiras, "Non-speculative store coalescing in total store order," in *45th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2018, pp. 221–234.

[15] C. A. Zukowski and S.-Y. Wang, "Use of selective precharge for low-power content-addressable memories," in *1997 Int'l Symp. on Circuits and Systems (ISCAS)*, Jun. 1997, pp. 1788–1791.

[16] G. Kucuk, K. Ghose, D. V. Ponomarev, and P. M. Kogge, "Energy-efficient instruction dispatch buffer design for superscalar processors," in *2001 Int'l Symp. on Low Power Electronics and Design (ISLPED)*, Jul. 2001, pp. 237–242.

[17] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006.

[18] 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.

[19] I. Park, C. L. Ooi, and T. N. Vijaykumar, "Reducing design complexity of the load/store queue," in *36th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2003, pp. 411–422.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

[27] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov. 2011, pp. 694–701.