Speculative Inter-Thread Store-to-Load Forwarding in SMT Architectures

Josué Feliu[a,*], Alberto Ros[a], Manuel E. Acacio[a], Stefanos Kaxiras[b]

[a]*Computer Engineering Department, University of Murcia, Murcia, 30100, Spain*
[b]*Department of Information Technology, Uppsala University, Uppsala, 752 36, Sweden*

Abstract

Applications running on out-of-order cores have benefited for decades of store-to-load forwarding which accelerates communication of store values to loads of the same thread. Despite threads running on a simultaneous multithreading (SMT) core could also access the load queues (LQ) and store queues (SQ) / store buffers (SB) of other threads to allow inter-thread store-to-load forwarding, we have skipped exploiting it because if we allow communication of different SMT threads via their LQs and SQs/SBs, write atomicity may be violated with respect to the outside world beyond the acceptable model of read-own-write-early multiple-copy atomicity (rMCA).

In our prior work, we leveraged this idea to propose *inter-thread store-to-load forwarding* (ITSLF). ITLSF accelerates synchronization and communication of threads running in a simultaneous multi-threading processor by allowing stores in the store-queue of a thread to forward data to loads of another thread running in the same core without violating rMCA.

In this work, we extend the original ITSLF mechanism to allow inter-thread forwarding from speculative stores (Spec-ITSLF). Spec-ITSLF allows forwarding store values to other threads earlier, which further accelerates synchronization. Spec-ITSLF outperforms a baseline SMT core by 15%, which is 2% better on average (and up to 5% for the TATP workload) than the original ITSLF mechanism. More importantly, Spec-ITSLF is on par with the original ITSLF mechanism regarding storage overhead but does not need to keep track of the speculative state of stores, which was an important source of overhead and complexity in the original mechanism.

Keywords:
Simultaneous multithreading, memory consistency, store-to-load forwarding, multiple-copy atomicity

1. Introduction

Synchronization and communication costs are one of the main scalability-limiting factors of parallel workloads. Despite the huge research effort devoted to make synchronization faster and more efficient (see, for example, [1], which provides a good review), it is widely known that synchronization strongly impacts on performance and scalability. Consequently, software researchers and developers seek to design more scalable algorithms, avoiding communication

*Correspdonding author. E-mail: josue.f.p@um.es

among threads as much as possible. This idea suits well many traditional parallel workloads where, basically, it is possible to distribute the data and work among threads at the beginning, let threads work independently for the largest fraction of the execution, and simply aggregate the results from all threads at the end. Traditional parallel benchmark suites such as SPLASH [2, 3] and PARSEC [4] are relatively synchronization-poor [5] and consequently their performance scales well in modern multicores. However, a different class of fine-grain, synchronization-intensive, parallel workloads has emerged recently. Workloads such as those that implement graph and tree algorithms [6] or write-intensive transaction processing [7] do not have straightforward synchronization-free versions and, consequently, cannot get rid of communication that easily because threads operate with shared data frequently.

Improving the performance of this kind of workloads is usually an exercise in frustration. Despite being parallel workloads, increasing the number of threads does not do much more than aggravate the synchronization problem: Because synchronization must be achieved via a common coherent level of the memory hierarchy, the farthest synchronization has to reach, the more expensive it becomes. Consequently, synchronizing two threads running in different processors via their shared main memory, is more expensive than synchronizing two threads running in the same processor through their shared last-level cache (LLC). And the latter is more expensive than synchronizing two threads running in a simultaneous multithreading (SMT) core via their shared L1. Therefore, consolidating threads that synchronize frequently in the same SMT core has potential to accelerate performance for fine-grain, synchronization-intensive, parallel workloads.

However, there is a greater chance to make communication faster within the SMT core: perform the inter-thread communication while instructions are still in-flight via store-to-load forwarding. The difference is high: communication through the L1 cache needs a store to be written to the L1 cache which in total store order (TSO) requires that the store is the oldest for that thread in the store buffer. Conversely, communication through the LQ and store queue (SQ) / store buffer (SB) could be potentially done as soon as the store computes its data and address.

Our inter-thread store-to-load forwarding (ITSLF) for SMT architectures proposal [8] is the first one that allows threads to communicate values to other threads before they are stored in the L1 cache and while they are still in the SQ/SB. ITSLF allows a thread to see stores from other threads in the same SMT core before other threads can see them. Obviously, this could have significant implications for the memory model. Specifically, it violates the acceptable read-own-write-early multiple-copy atomic (rMCA) model used, among other, by TSO. However, using speculation, ITSLF safeguards write serialization for same-address stores while they are only locally visible in the SMT (but not globally visible outside the SMT) and efficiently maintains rMCA both within and outside the SMT.

ITSLF restricts forwarding store values to other threads in the SMT core from the SQ/SB to non-speculative stores. Since these stores never get squashed, this design choice does not require the core to keep track of the inter-thread forwardings. However, restricting inter-thread forwarding only to non-speculative stores affects the hardware cost and performance of the mechanism. First, ITSLF requires keeping track of the speculative state of stores [3], which adds a non-negligible source of overhead and complexity. Second, it requires delaying the LQ search that stores trigger to their own LQ when a store executes (see Section 2.1) to the time the stores become non-speculative. And third, it delays the time when the store value can be forwarded to other SMT threads in the core, which constraints the performance benefit that inter-thread forwarding can provide.

In this paper, we extend the original ITSLF mechanism to allow inter-thread store-to-load

forwarding across SMT threads as soon as the stores compute their address and their data is available. In other words, we extend ITSLF to allow forwarding from speculative stores. Speculative Inter-Thread Store-to-Load Forwarding (or simply Spec-ITSLF) extends the core with a small Inter-Thread Forwarding Table to keep track of the inter-thread forwardings. Overall, Spec-ITSLF requires only 190 bytes for an 8-way SMT Ice Lake like processor, which is on par with the original ITSLF proposal (200 bytes). Spec-ITSLF not only improves the performance of the original ITSLF mechanism but, more importantly, it strongly simplifies its implementation on top of a baseline SMT processor. Particularly, Spec-ITSLF gets rid of the hardware required to keep track of the speculative state of stores and keeps the LQ search performed by stores when stores compute their address, as it is done in a baseline SMT core.

2. Background

Simultaneous Multithreading (SMT) is the only multithreading paradigm that allows issuing instructions from different threads in the same cycle. This capability increases the chances of finding ready instructions to be issued each cycle, therefore improving the core utilization and throughput. To achieve these benefits while keeping the overhead of implementing SMT low, SMT cores share most of their resources among the threads co-running in the same SMT core. Figure 1 shows how the different resources of the pipeline of an SMT core are shared in an SMT implementation that resembles Intel's implementation of SMT [9].

A design choice to minimize the overhead of implementing SMT is to time-share the fetch, decode, rename, dispatch, and commit stages among threads so that they operate with a single thread each cycle (as in a non-SMT core). Intel follows this approach in their SMT cores [9] and is the one we assume in this work (see Figure 1). Only few resources need to be replicated for each thread (e.g., the program counter, the Register Alias Table, the return stack) and the size of the physical register file grows to account for the increase in the architectural registers while leaving the same number of physical registers to hold renamed state.

The rest of the physical resources of the baseline non-SMT core are *shared* among threads *without increasing their size*, as also done by Intel in its SMT processors [9]. These resources include: the execution units, the reorder buffer (ROB), the load queue (LQ), the store queue (SQ) and the store buffer (SB). Besides the execution units that form a common pool for all threads, sharing of a physical resource creates multiple smaller *logical* copies of the resource, one for each thread (see Figure 1). This is accomplished either *dynamically* by using thread_ID tags to discriminate its entries, or *statically* by physically partitioning the resource to the different threads. We take the second approach but this choice is orthogonal to our proposal.

Finally, we distinguish between the SQ and the SB: the SQ contains stores that may have been executed but not yet committed, the SB contains stores that have committed but have not yet been performed (written in the L1), i.e., not yet inserted in the global memory order. In some implementations, the SQ and SB are the same physical structure (circular FIFO queue) and the distinction between them exists only via a pointer that marks the entries belonging to the SQ and to the SB respectively [10]. This implementation is orthogonal to our approach.

Atomic instructions typically empty the SB to execute and commit [11]. In SMT, an atomic instruction empties the SB of its own thread before executing but has no effect on other SBs.

2.1. Speculative support for memory ordering

Today's cores issue memory operations speculatively out-of-order. Correctness is ensured in presence of out-of-order execution by checking that (1) memory dependencies and (2) load→load

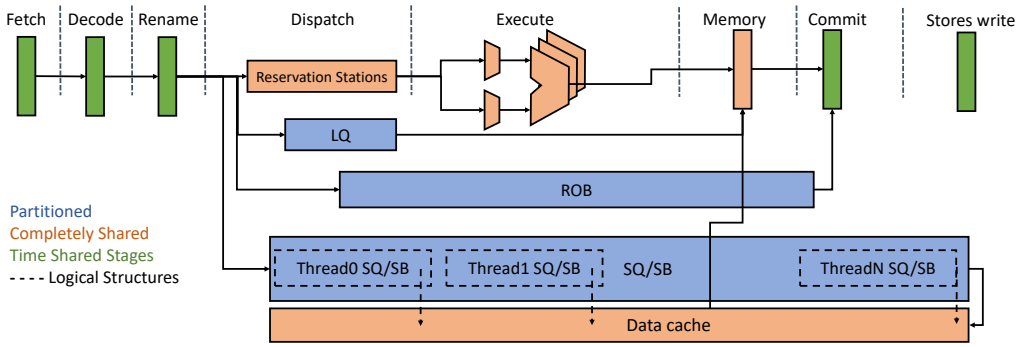


Figure 1: Sharing of the pipeline resources among threads in a typical SMT model.

ordering are respected. These checks require frequent associative searches on the LQ and the SQ/SB. The LQ and SQ/SB are implemented as CAMs (content addressable memories) and are among the most expensive processor structures. More importantly, these structures pose a critical trade-off: On one hand, their size (in number of entries) should be increased enough to prevent stalls due to capacity limitations. This is especially important in a market environment where newer generations of commercial processors increase the number of in-flight instructions (see, for example, Intel Ice Lake [12] or Apple's M1 [13]). On the other hand, these CAM structures need to be searched fast, which limits their size (or alternatively, larger size makes them slower).

Memory dependencies are respected when loads read the latest value written by a store in the same thread to the same address, if no other thread wrote that location in the interim. However, out-of-order processors speculatively issue loads over older stores that have not been performed (written to cache), or even not issued i.e., have not computed their target addresses. Overall, to respect memory dependencies, three types of CAM searches, detailed in the paragraphs below, are needed: i) loads must search the SQ/SB; ii) stores must search the LQ; and iii) external stores, that manifest as invalidations reaching the core, must also search the LQ.

Loads searching the SB: To retrieve the data from committed but non-performed stores, the SB[1] is searched by every load, in parallel to the access to memory. On a hit in the SB, the store *forwards* the data to the load. To maintain the highest performance, a parallel search of the SB implies that the SB: i) should have at least the same number of ports as the L1 cache has for read operations (usually two); ii) should be searched with a latency not larger than the L1 latency, so as to not incur a penalty on hits; and iii) should be segmented, to allow executing additional search operations per cycle. A recent study reports that no fewer than four cycles are needed to forward data from a store to a load in an Intel Skylake, and no fewer than five cycles in an Intel Ice Lake [14].

Stores searching the LQ: Since there may be stores with unresolved addresses when a load snoops the SQ/SB, *every store* needs to snoop back the LQ once it computes its address. Loads that have executed in the presence of an older unresolved store, are called ***Data-Speculative (D-Speculative)***. When a store snoops the LQ, if there is a match with a younger D-Speculative load, the load and subsequent instructions are *squashed and re-executed* as the D-Speculative

[1]We include here, for the same thread, the stores (older than the load) that may still be in the SQ.

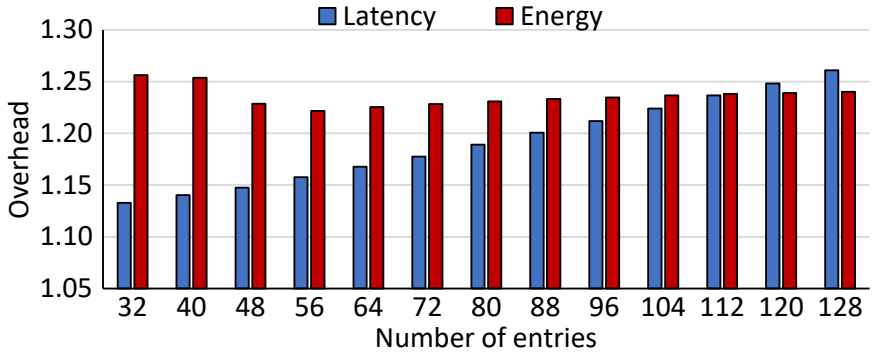


Figure 2: Search latency and energy consumption overheads of adding a second search port to the LQ (CACTI-P results).

load breaks a memory dependence. This is a rare event due to accurate memory dependence prediction [15] that exists today in most architectures, that prevents *memory dependent* younger loads to execute in the presence of the unresolved older stores. In processors where a single store is issued per cycle, the LQ can implement a single search port, allowing it to be larger than the SQ/SB with a similar latency. Interestingly, CACTI-P [16] reports similar search latency for a 128-entry CAM with 1 port than for a 72-entry CAM with 2 ports, sizes of the LQ and the SQ/SB, respectively, of an Intel Ice Lake processor [12].

External stores searching the LQ: *Load→load* order is required for loads to the same address to guarantee coherence [17], and for loads to different addresses in systems that provide strong consistency guarantees [18], such as x86-TSO [19]. To respect the *load→load* order semantics, the LQ is searched when receiving an invalidation, as another core's write may be exposing a speculative load reordering. Cache evictions are conservatively treated as *potential invalidations* (also searching the LQ) as any actual invalidation would never reach the LQ in this case. If a match occurs, the load that is caught violating the *load→load* order, called ***Memory-Speculative (M-Speculative) load***, and all subsequent instructions are squashed and re-executed. Since these searches are not frequent, it is preferable not to add an extra search port to the LQ, but just perform the searches when the LQ port is free, potentially delaying invalidations or evictions. As shown in Figure 2, adding an extra port increases search latency, especially for queues with a large number of entries, risking a negative impact on performance. Energy consumption of the LQ is negatively impacted as well.

2.2. Speculative support for memory ordering in SMT architectures

Consider, now, an SMT core where each hardware thread only "sees" its own *logical* LQs and SBs. This is currently how SMT implementations work and we will explain in the next section what mandates this behavior. In this case, a speculative reordering that violates *load→load* order in one thread could be exposed by stores performed by a second hardware thread *in the same core*. However, *coherence* invalidations to the LQ of the first thread, emanating from the stores of the second thread, *are not forthcoming by default* as both threads share the same coherent state of the cachelines in the L1.

A naïve solution is to force the behavior of an invalidation by performing an LQ search on each of the threads in the core whenever any store is written from a SB to the cache. This LQ

search can be performed in parallel to the store writing to cache. On a match, on any LQ, the matching speculative load and the subsequent instructions of the corresponding thread should be squashed. Note that, in contrast to invalidations where the whole cacheline range of addresses is searched in the LQ (as invalidations work at the granularity of a cacheline), stores just search for loads matching the exact address that they write, thus removing false-sharing effects. Alexander et al. [20] focus on an SMT processor with a strongly ordered consistency model and propose to trigger byte-precise LQ and SQ searches for each executed load and store, respectively, to avoid potential consistency violations. As discussed in the previous section, adding a second port to the LQ negatively impacts search latency. Using the existing port for inter-thread searches, may also negatively impact performance too, as inter-thread searches are far more frequent than external invalidations or even evictions.

2.2.1. LQ snoop filtering in SMT architectures

The naïve solution searches the LQs of the other SMT threads *on every store*. A possible optimization is to filter LQ searches by adding "LQ directory" information to the L1 cachelines, to track whether any other thread is reading a cacheline [21]. When a store is written from the SB to the L1, it checks the LQ directory of the cacheline. If a different thread has read the cacheline since the last time it was written, the LQ of that thread is searched looking for a matching load to squash, and the thread is marked as not reading the cacheline anymore. We can view that information as a "need LQ search" bit per hardware thread [21]. If no other thread has read the cacheline, no LQ search is needed. In case of a cache miss, no LQ-directory information exists, but no LQ snoop is required as any LQs were already searched, if required, when the cacheline was last evicted.

The LQ directory approach significantly reduces the number of LQ searches when stores write to memory, as many cachelines are not shared by different threads. Consequently, it reduces contention in the LQ snoop port and saves energy. However, this approach increases the complexity of cache writes. Since the LQ-directory information has to be retrieved from the cacheline tag, the LQ snoop cannot be initiated until the L1 access is performed. We give advantage to the filtering baseline with an idealized implementation and optimistically assumes that reading the bitvector and determining if an LQ snoop is needed does not lengthen tag lookup and thus the write still performs in the same L1D access latency. If an LQ snoop is indeed needed, the write is immediately relaunched in parallel with the LQ snoop and complete together. The cache latency for such a write doubles. Other, already started, writes to the L1 are optimistically delayed just enough to prevent store-store ordering violations within the same thread. Because of this, and despite our idealized implementation, the LQ-directory solution is *not consistently better* than the baseline, leading in some cases to performance degradation as we show in Section 5. In addition, storing the LQ-directory information along the L1 cachelines, also incurs an overhead of N bits per cacheline, with N equal to the number of supported SMT threads in the core.

3. Issues and Solutions with Spec-ITSLF

Allowing inter-thread store-to-load forwarding from a thread to another in the same core in an SMT architecture has the potential to accelerate communication between threads. Inter-thread forwarding can be simply enabled by not restricting the SQ/SB search performed by loads to just the stores belonging to the same thread. SMT processors already include hardware support to perform this search when configured in single-thread mode.

However, as stated by Nagarajan et al. [11], a thread cannot read a value written by another thread on the same core before the store has been made "visible" to threads on other cores (i.e., globally ordered). This implies that a thread cannot get the value forwarded from another SQ/SB in the same core, but it has to wait until the store is inserted in memory order. As we show here, the reason is that allowing inter-thread forwarding exposes store values before they are inserted in global order, not just to loads from the same thread, but also from other threads in the same core. This breaks: i) coherence, ii) TSO, iii) write serialization, and iv) rMCA, which is respected by most vendors (e.g., x86-TSO [19], ARMv8 [22]), resulting in a more complex non-MCA model where stores are not globally ordered. To the best of our knowledge, our previous work [8] is the first discussion in the literature about the coherence/consistency impact of inter-thread forwarding in SMT.

This work extends the discussion considering inter-thread forwarding from speculative stores. When allowing speculative stores to forward to other threads in the core, a new scenario arises: The forwarding store could be squashed, leaving the forwarded loads with wrong data because, in a baseline SMT, squashing an instruction from a thread only affects the younger instructions from that thread. Next, we extend the original discussion to explain the new situations that arise and how Spec-ITSLF addresses them to guarantee rMCA.

3.1. Point of Local Visibility

In the single thread case, stores that resolve their address squash D-Speculative younger loads on the same address that have executed speculatively, bypassing the unresolved-address store. A store, by squashing such younger D-Speculative loads, ensures that it will be the one visible to all of them when they re-execute. More importantly, stores make their presence known to other threads when they write to memory via invalidations that search the LQs of other threads (in other cores) to squash speculative loads that may be violating memory model semantics. If we allow ITSLF[2] in an SMT core, we lack an analogous mechanism to prevent scenarios such as the coherence (Figure 3(a)) and TSO (Figure 3(b)) problems, presented below.

Single-Address Coherence Example. Consider the simple *coherence* problem depicted in Figure 3(a), the same example as in Dubois et al. [17]. Note that this example applies to every memory model. The value in between parenthesis in the loads is the value read by them. The number before each instruction indicates the memory order for the depicted execution. This execution could happen in single-thread cores, or in SMT, and the solution is always the same (search LQ and squash – see below). Assume that the loads in this example are speculatively reordered. The second load performs before `st x, 1` is visible to thread 1 and reads the value 0, creating a from-read happens before dependence with the store. Then `st x, 1` computes the address and becomes a potential forwarder. Finally, in an SMT with ITSLF, the first load executes getting the value (1) forwarded from the store, creating a read-from happens before dependence with the store. A dependence cycle is created when considering program order and this execution breaks the *coherence expectations* for variable x.

TSO Example. A similar problem appears also when we have multiple addresses and forwarding. Consider the mp litmus test shown in Figure 3(b). In this example TSO is violated by ITSLF. If initially `x, y = 0`, then getting in thread 1 `x == 1` and `y == 0` is not allowed by TSO.

[2]Throughout this section we use the term ITSLF to refer to inter-thread store-to-load forwarding in a general way, without distinguishing whether the forwarding store could be speculative (Spec-ITSLF) or must be non-speculative (original ITSLF). When the difference between both techniques is important, we explicitly differentiate between the Spec-ITSLF and (original) ITSLF mechanisms.

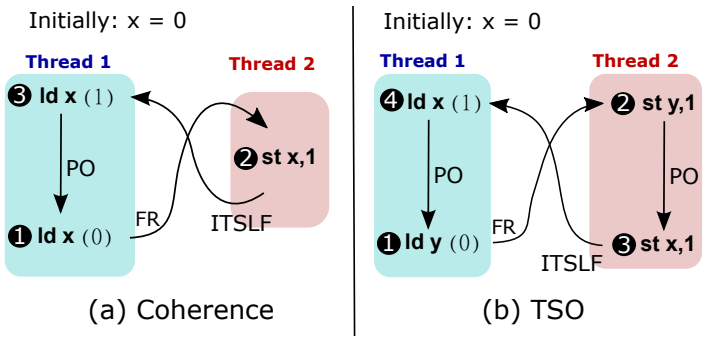


Figure 3: Coherence (a) and TSO examples (b).

Imagine that thread 1 executes `ld y` out-of-order before `ld x`. Thread 2 has not issued any store yet. Clearly `ld y` reads speculatively the value 0. Now, thread 2 executes (computes the target address) `st y, 1` and `st x, 1` so they are visible to other threads in the core. Now, `ld x` executes, getting the value (1) forwarded from `st x, 1`. But this would create a cycle when considering the program order, thus breaking TSO.

Spec-ITSLF Solution. We define a store to be *locally visible* when its value can be forwarded to another thread in the same core. The three fundamental requirements for forwarding are (1) the store address has been computed, (2) the store value is available, and (3) the store is still in the SB (not inserted in global order yet).

To fix both the coherence and TSO problems, we combine the single-thread store-to-load forwarding and the external invalidation approaches of the baseline SMT in a single ITSLF approach: *A store must search the LQs of other threads, and squash the matching M-speculative loads, in order to become visible to these threads.* Because the store is not ordered in relation to the instructions of other threads, it squashes any matching M-Speculative load (same address) without having a concept of "younger." But to its own thread, the store behaves normally and squashes only younger matching D-Speculative loads.

In the baseline SMT, a store searches the LQs of other threads only after it writes to the L1 and becomes globally visible. *This is equivalent to an external invalidation due to a store of another core.* The question is: at what point do we allow a store to squash loads in ITSLF? There are two choices. If we allow forwarding to other threads for *non-speculative stores*, either retired in the SB or bound-to-retire in the SQ, the stores should search other LQs at the point when they become non-speculative. This is the design choice taken for the original ITSLF proposal [8]. A key realization to make both local thread store-to-load-forwarding and ITSLF work seamlessly together, *in a single LQ snoop*, is that: *it is always correct for a store to wait until it is ready to commit, in order to perform the squash to its own-thread younger D-Speculative loads.* At that point the store combines its local LQ squash with the squash of other thread M-speculative loads in other LQs.

With Spec-ITSLF, we go a step further and allow forwarding from the point stores compute their address. Note that, in the same thread, younger loads always *see* the thread's own stores from this point of time. Allowing stores to forward to any SMT thread in the core from this point brings three important benefits. First, it allows forwarding the data earlier, without waiting until the store becomes non-speculative. Second, it allows merging both LQ squashes (same

thread and other threads) into a single one triggered when stores compute their address, the same time when the baseline SMT core triggers its local LQ squash. In contrast, to merge both LQ squashes, the original ITSLF mechanism requires delaying the local LQ squash until the store becomes non-speculative. And third, it allows eliminating the additional hardware required by the original ITSLF to keep track of the speculative state of the stores. With Spec-ITSLF, we arrive at the following invariant:

Invariant: *Stores become locally* visible *to SMT threads when they compute their address and the store data is available (as in same-thread store-to-load forwarding). At that point, they squash the younger matching D-speculative loads in their own thread and any matching M-speculative load in all other threads. When a store becomes locally visible it can forward its data to loads of any thread in the SMT core.*

It is now straightforward to see that in the coherence example (Figure 3(a)), when `st x, 1` becomes visible, it triggers an LQ snoop that squashes `ld x (0)`, breaking the dependence cycle and ensuring that the next time the load executes, it reads the new value. Similarly, the dependence cycle is broken in the TSO example (Figure 3(b)) when `st y, 1` becomes visible.

Establishing a unique point of squash for a store, when it becomes locally visible, does not incur any additional cost over the baseline: a store still snoops, a single time, the same total number of LQ entries in the single-thread-baseline (ST-baseline) or in SMT mode (the thread LQs in SMT add up to the single LQ in the ST-baseline).

Allowing stores to forward to any thread in the core from the point they compute their address, however, enables inter-thread forwarding from speculative stores (e.g., due to a branch prediction) that could be squashed later. Therefore, to guarantee correctness, Spec-ITSLF should propagate the squash of a store to the forwarded loads from all SMT threads in the core. As we discuss later, loads forwarded from a different thread are not allowed to commit until the forwarding store writes to the L1 and remain speculative until that time, which prevents loads forwarded from a speculative store of a different SMT thread from committing with a *wrong* value. Section 3.4 describes the implementation details of Spec-ITSLF and discusses how it keeps track of the inter-thread forwardings and ensures the propagation of store squashes to all forwarded loads in an efficient way.

3.2. Local Store Order

Establishing a point of local visibility for each store is not enough to solve a separate problem: write serialization (two stores to the same address by any two threads are observed in the same order by all threads). Consider the example below.

Write Serialization. In Figure 4(a), both stores are locally visible. Assume that `ld x (2)` executes and reads 2. Then, `st x,2` performs and exits the SB. `ld x (1)` reads 1. Finally `st x,1` performs and the memory is left with the final value of 1. The problem is that the SBs of threads 2 and 3 are not ordered, and if thread 3 writes to cache first, we have the IRIW problem (two observers do not agree about the order of the stores—assume, for example, a coherent observer in another core).

Spec-ITSLF Solution. The problem here is that for the same address we need to decide which store is *younger*. Same-address stores in the same SB are either ordered (TSO) or coalesced (relaxed models). The effect is the same: only the younger store forwards. But across the SQs/SBs of multiple threads no relative order exists for locally visible same-address stores. Worse: the global order is established only when the stores are written in the L1 and it is irrevocable after that. (In the SMT model we use, we allow the heads of the SBs to be written in the

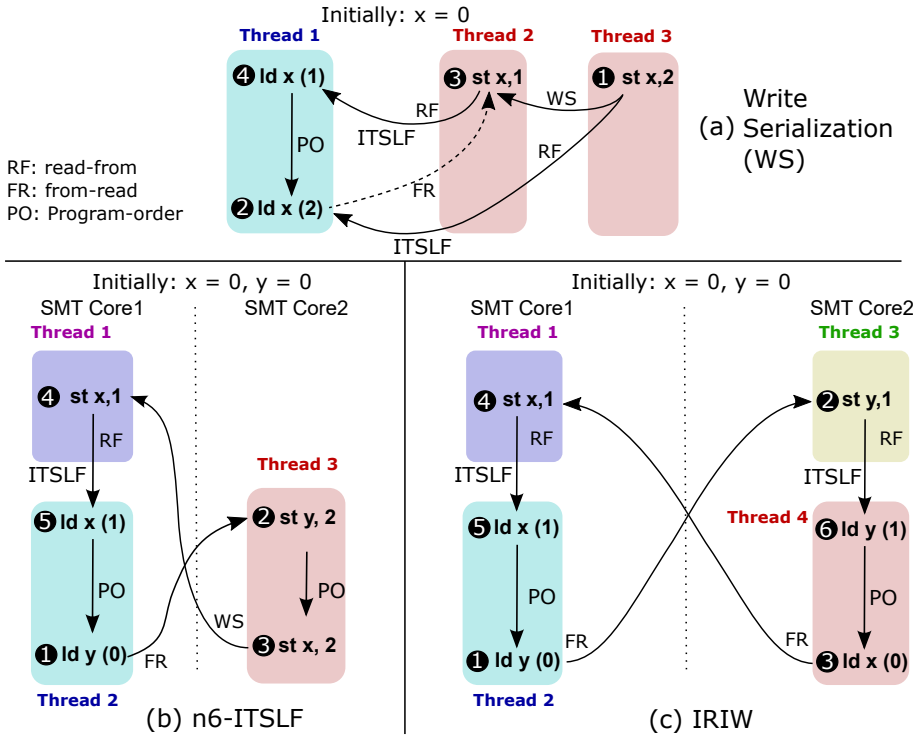


Figure 4: Write serialization (a), n6-ITSLF (b), IRIW (c).

L1 in arbitrary order.) This means that it is impossible to decide on a local order without first knowing the global order.[3]

To solve this problem, we allow only one store (of a particular address) to forward to loads based on *local visibility order (LV order)—or LQ snoop order*. When a load snoops all SQs/SBs and matches several candidates, in more than one SQ/SB, these candidates are ordered by their LV order. Right at that point the load selects the "youngest-to-LV" store for forwarding. Note that similar functionality already exists in the ST-baseline unified FIFO SQ/SB (for TSO): all stores of the same address are matched by a load and the youngest store is selected to do the forwarding. In the ITSLF case, a load can match multiple stores in multiple SQs/SBs (that all add up to the ST-baseline SQ/SB). We select the youngest-to-LV.

Unlike the original ITSLF, Spec-ITSLF allows inter-thread forwarding from speculative stores. This adds the case where a speculative store forwards to loads from other SMT threads and, after that, gets squashed. As described earlier, to guarantee correctness, Spec-ITSLF propagates the store squash to the loads from any thread that got the data forwarded from the squashed store. Note that this action is enough to guarantee the local store order in Spec-ITSLF. The squashed loads will execute again and will get the data forwarded from the youngest store based

[3]In addition, there is a deadlock danger if we try to establish a local order for more than one address that turns out to be the opposite order in the global order.

on the local visibility order at that time.

Therefore, we simply need to extend the SQ/SB entries with a field to store their LV order. The cost of this field is quantified in Section 3.5.

Now, combine this approach with the *Invariant of the Point of Local Visibility*, discussed above: Whenever a store becomes visible (computes its address and has the store data available), it snoops the LQs of the other threads and squashes all speculative loads on the same address that may have forwarded from older-to-LV stores. The store becomes the youngest-to-LV and prevents all older-to-LV stores from ever forwarding again while they are in the SQ/SB.

Invariant: *Only a single store on a particular address, the youngest-to-local-visibility (youngest to become visible), can forward to loads.*

At this point the reader may be concerned that the LV order of the stores may be different from their eventual global order. This can be true but it does not matter. As we will see next, the key idea that puts everything together (solving the multiple address and rMCA problems) is that the loads that "see" a store (through forwarding) are obliged to commit only after the store is inserted in the global order (written from the SB to the L1) and must remain speculative (exposed to squashing) until that time.

3.3. Multi-Copy Atomicity

Finally, we address the main culprit that prevents ITSLF in current systems: violation of MCA. Informally, in memory models that demand MCA, all threads should see stores in global memory order at the same time. In an SMT with ITSLF, local threads can see each other's stores even if these stores have not been inserted in the global memory order, i.e., are still in their SQs/SBs. Obviously, this would violate MCA in SC, TSO, or even in relaxed memory models.

In recent work, Ros and Kaxiras [23] show that a detection of an MCA violation that stems from store-to-load forwarding, appears as loads observing stores in a different order. Of course, this behavior, if it stems from a same-thread store-to-load forwarding, is incorporated in memory model definitions such as x86-TSO [19] or ARMv8 [22] and is known as *read-own-write-early multiple-copy atomicity, (rMCA)*. In other words, stores appear at the same time to all threads, except to the own thread where they might appear earlier (before inserted in the global order). However, *if the forwarding is from another thread then the same behavior would be a violation of rMCA*.

It is straightforward to show using two classic litmus tests that ITSLF leads to violations of rMCA. More specifically, the cycles that appear in n6-ITSLF (a variation of n6 [19], Figure 4(b)) and IRIW [24] (Figure 4(c))—discussed below—are not due to read-own-write-early —they are forwardings from other threads— and, therefore, violate rMCA.

n6-ITSLF Litmus Test. Consider the *n6-ITSLF* litmus test running in Thread 2 and Thread 3 in Figure 4(b). The difference with n6 is that the store-to-load-forwarding is ITSLF. In x86-TSO it is not possible this outcome: `[x]==1; [y]==2; x==1; y==0;` since that would create a cycle by allowing Thread 2 to see the store of Thread 1 before that store is globally ordered with respect to Thread 3. Seeing this cycle would mean that either the loads or the stores are reordered, or alternatively, rMCA is not respected, i.e., the system is non-MCA. Executing the first two threads in the same SMT core, and allowing ITSLF obviously allows this to happen.

IRIW Litmus Test. Similarly, ITSLF also breaks the Independent Reads Independent Writes (IRIW) litmus test by creating a cycle, allowing local threads to see stores earlier than remote threads, thus violating rMCA. The cycle means that two independent stores (writes) cannot be ordered which is not (generally) true from the point of view of an outside observer.

Spec-ITSLF Solution. When a load gets the data forwarded from a store of a different SMT thread in the core, it is not allowed to commit until the forwarding store writes to memory. Therefore, a load at the head of the ROB, checks if SQ/SB still contains the forwarding store, or otherwise, the store has been written to L1. In the first case, the load will not be committed. Section 3.4 presents the implementation details and discussed how this check is efficiently implemented.

Invariant: *A load receiving forwarded data from a different thread: i) cannot retire from the ROB (commit) until the forwarding store becomes globally visible;* and *ii) until it retires, the forwarded load makes all younger loads in its thread* store-atomicity-speculative*, therefore subject to squashing from conflicting stores.*

Based on the previous invariant, in the *n6-ITSLF* litmus test, ITSLF does not allow `ld x (1)` to retire until `st x, 1` does, and so it remains speculative and is squashed when `st x, 2` is made visible. Similarly, in the IRIW litmus test, ITSLF does not allow `ld x (1)` (thread 2) and `ld y (1)` (thread 4) to retire until their forwarding stores do, leaving `ld y (0)` and `ld x (0)`, respectively, exposed to squashes due to invalidations.

Interestingly, the previous invariant also solves the two issues that arise in Spec-ITSLF when a speculative store that forwarded to loads from other threads gets squashed. First, because loads forwarded from a different SMT thread should wait until the forwarding store performs to commit, it is not possible that a load forwarded from a different thread commits with a wrong value if the forwarding store eventually gets squashed. And second, because the squash of a store is propagated to the forwarded loads from all threads, these loads will re-execute and get forwarded from the youngest-to-local-visibility store at that time, respecting the defined local store order.

3.4. Spec-ITSLF implementation details

Spec-ITSLF needs to keep track of the inter-thread forwardings to ensure that (i) loads forwarded from a store of a different SMT thread in the core do not commit until the forwarding store writes to L1, and (ii) squashed stores propagate the squash to forwarded loads from all SMT threads in the core when needed. To keep track of the inter-thread forwardings in a efficient way, Spec-ITSLF employs a small hardware structure called the Inter-Thread Forwarding (ITF) table. Figure 5 shows the fields that form an ITF entry: a valid entry bit (`V`), a forwarding `Store id` (i.e., the position of the forwarding store in the SQ/SB), and for each SMT thread in the core, a valid load id bit (`v`) and the `Load id` (i.e., the position of the forwarded load in the LQ).

Keeping track of inter-thread forwardings. When a load matches a different thread store and it is to be forwarded, the forwarding should be recorded in the ITF table. If the store did not forward to loads from other threads before, a new entry in the ITF table should be allocated, setting the valid bit in the entry and saving the store id and load id in the corresponding fields. If an entry in the ITF table is needed but the table is full, the forwarding is cancelled and the load is re-scheduled. The SQ/SB entries (Figure 6a) are extended with a field to store a pointer to its assigned entry in the ITF table (`ITF entry id`) and its valid bit (`V`) that determines if the previous field holds a valid pointer. [4] With these fields, we avoid associative searches in the ITF table. If the store already forwarded to loads from other threads, it should have an ITF entry assigned and a pointer to this entry stored in the SQ/SB entry. In this case, the load id is saved

[4] As discussed in Section 3.2, SQ/SB entries are also extended with the `LV order`, used to determine the local visibility order of stores.

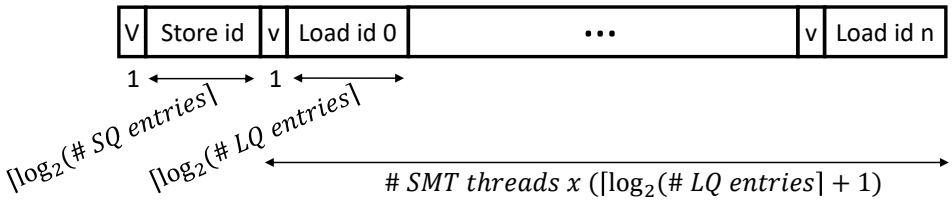


Figure 5: Inter-Thread Forwarding (ITF) entry.

in the corresponding field of the ITF entry unless there is an older load from the same thread saved (identifying which load is older is straightforward using the loads ids). Finally, the LQ entries (Figure 6b) are also extended to save a pointer to the corresponding entry in the ITF table (`ITF entry id`) and a valid bit (`V`) associated to the pointer. When a forwarded load replaces a younger forwarded load from the same thread in the ITF entry, it uses the pointer to the younger load's LQ entry to clear the validness of its pointer to the ITF table.

Stores free their ITF entry when they write to memory or when they are squashed. In the former case, only the valid bit in the ITF entry should to be cleared. In the latter case, the squash is propagated to the oldest load of each thread that read the data forwarded from this store.

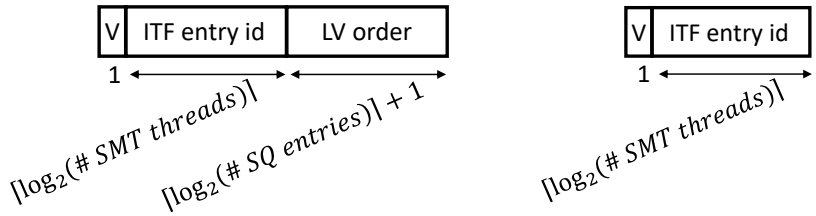
Propagating store squashes. If a store that forwarded to other SMT threads is squashed, it checks its ITF entry, squashes the oldest forwarded load and all younger instructions from each thread, and clears the valid bit in the ITF entry. This ensures that the loads will read the correct value when they re-execute. Since a load forwarded from the store could have been executed speculatively and squashed, squashing a load requires checking if the forwarded load is still present in the LQ entry or if the entry currently holds a different load. If the load's LQ entry holds a valid pointer to the ITF entry of the squashed store, the load is still present and should be squashed.

Notice that it is not possible for forwarded loads to commit after reading the *wrong* value from a store that is to be squashed later since a load that is forwarded from a different thread store is not allowed to commit until the forwarding store is performed (i.e., writes to the L1).

Guaranteeing MCA. A load at the head of the ROB with a valid pointer to its forwarding store entry in the ITF table checks if the forwarding store entry in the ITF table is still valid and if the store currently holding the entry forwarded its data to this load (i.e., the load id in the ITF entry matches the load's position in the LQ). (These checks are needed because the ITF entry could be freed and re-used by a different store before this load had the chance to commit.) If the ITF entry pointer is valid and the forwarded load id matches the position of the load in the LQ, the forwarding store is still to be performed and the load should not be committed yet. Otherwise, the forwarding store has been written to L1 and the load is allowed to commit.

3.5. Hardware Cost

Inter-thread store-to-load forwardings are tracked of in the ITF table comprised of the entries depicted in Figure 5. The `Store id` requires $\lceil \log_2(\text{SQ/SB entries}) \rceil$ bits. Each `Load id` requires $\lceil \log_2(\text{LQ entries}) \rceil$ bits. Overall, an ITF entry requires $1 + \lceil \log_2(\text{SQ/SB entries}) \rceil + \text{SMT threads} \times (\lceil \log_2(\text{LQ entries}) \rceil + 1)$ bits. In an Ice Lake core with a 72-entry store buffer and a 128-entry load queue, and assuming an 8-way SMT core, each entry requires 72 bits. As Section 5.1



(a) SQ/SB entries. (b) LQ entries.

Figure 6: Fields added to the SQ/SB and LQ entries to support Spec-ITSLF.

discusses, few entries in the ITF table are enough to provide the highest benefits. Thus, we build the ITF table with just 2 entries, which makes the ITF table only 18 bytes.

We extend the SQ/SB entries with a field to store their LV order, a field to refer to the corresponding ITF entry when the store forwards to loads from other SMT threads, and a bit to indicate if the previous field holds a valid pointer to the ITF table. The `LV order` field requires $\lceil \log_2(\text{SB entries}) \rceil + 1$ (sorting-bit). The sorting bit [25] is a low-cost implementation of a monotonically increasing number that uses the SQ position augmented with an extra bit called "sorting" bit to handle wrap-around.[5] The pointer to the ITF entry requires $\lceil \log_2(\text{SMT threads}) \rceil$ bits. Assuming a core with a 72-entry store buffer, each SQ entry requires 12 additional bits (8 bits for the LV order, 3 bits for the pointer to the ITF entry, and the valid bit). Overall, the fields added to the SQ/SB require 108 bytes.

Finally, we extend the LQ entries with a field to refer to the corresponding ITF entry when the load is the oldest load forwarded by a particular store of a different SMT thread and a bit to indicate if the previous field holds a valid pointer to the ITF table. The pointer to the ITF entry requires $\lceil \log_2(\text{SMT threads}) \rceil$ bits. Assuming a core with a 128-entry LQ, the LQ is extended with 64 bytes (4 bits per LQ entry).

Overall, the hardware overhead of implementing Spec-ITSLF is only 190 bytes. That is 10 bytes less than the original ITSLF mechanism to additionally support forwarding from speculative stores. Furthermore, Spec-ITSLF does not require the hardware support to keep track of the speculative state of stores (e.g., [26]) required by the original ITSLF, which was an important source of overhead and complexity in the original proposal.

3.6. Summary

Table 1 summarizes the main actions performed by the ST-baseline, SMT-baseline, (original) ITSLF, and Spec-ITSLF along the different execution steps of loads and stores. Two key differences contribute to make an SMT core with ITSLF support better than the baseline SMT. First, when loads execute, they search the SQ/SB of all threads and read the data from the youngest store among same-thread stores and other-thread locally visible stores in the SQ/SB. Second, ITSLF merge the two LQ snoops that stores perform (the first one, when they execute, to squash

[5] As stores become non-speculative and snoop the LQ, their LV order is set appending a sorting bit of 0 to an order counter of $\lceil \log_2(\text{SB entries}) \rceil$ bits and, after that, the counter is incremented. When the counter wraps around to value 0, all sorting bits are set to 1 in the SQ. These steps guarantee that new stores that snoop the LQ commit will have a lower LV order than *older* stores that snooped the LQ earlier.

Table 1: Summary of the actions performed by the ST-baseline, SMT-baseline, ITSLF, and Spec-ITSLF for load & store execution. The red color highlights the associative searches to the LQ and SQ/SB. The green color shows when stores become locally visible to other threads running in the SMT core in the (original) ITSLF and Spec-ITSLF mechanisms. Finally, the blue color indicates actions performed by Spec-ITSLF related the ITF table.

	ST-baseline	SMT-baseline	ITSLF	Spec-ITSLF
LD exec	**Search SQ/SB.** Forward data from most-recent matching ST.	**Search (own-thread) SQ/SB.** Forward data from same-thread most recent-matching ST. ***Read locked lock from L1 when other thread is about to free it.***	**Search SQ/SB.** Forward data from the (youngest-to-LV) ST in SQ/SB (all threads). ***Read other thread freeing the lock directly from the SQ/SB!***	If inter-thread forwarding: record it in the ITF table.
LD retire			**If forwarded, wait for ST to perform.** (A load forwarded from a different thread cannot retire until the store writes.)	
ST exec	**Search LQ.** Squash matching younger LDs.	**Search (own-thread) LQ.** Squash same-thread matching younger LDs.	Search-LQ functionality deferred to the moment the store becomes non-speculative and bound-to-retire.	**ST becomes visible** to all threads in the core. **Search (all threads) LQ.** Squash matching speculative LDs from any thread (only younger from own thread).
ST becomes non-speculative	–	–	**ST becomes visible** to all threads in the core. **Search (all threads) LQ.** Squash matching speculative LDs from any thread (only younger from own thread).	–
ST squashed				Propagate squash to all forwarded loads through the ITF table. Clear the store ITF entry.
ST performed	**Write L1.**	**Write L1.** **Search (all threads, except own) LQ.** **(Filtering:** Search only if other threads share the cacheline.) Squash matching LDs from other threads.	**Write L1.** (Forwarded load(s) waiting the store to write will now be able to retire.)	Clear the store ITF entry.

same-thread D-speculative loads and, the second one, when they write to memory, to squash M-speculative loads from other threads) into a single one. This reduces LQ snoop port contention and helps improve the SMT performance when running synchronization-poor workloads. The key differences between the Spec-ITSLF and (original) ITSLF mechanisms are that Spec-ITSLF allows inter-thread forwarding from speculative stores and performs the LQ search when the stores compute their address. Conversely, the original ITSLF only allows inter-thread forwarding from non-speculative stores and defers the LQ search until the stores become non-speculative.

ITSLF does not increase the amount of work done by the baseline single-threaded or SMT cores significantly. In terms of CAM searches, stores search the entire LQ during their execution in one LQ search (ITSLF and the single-threaded baseline) or in two LQ searches (SMT baseline) and loads search the entire SQ/SB in ITSLF and the single-threaded baseline. While loads could search only the own thread SQ/SB segment in the SMT baseline, selectively searching partitions of the LQ requires additional logic that might not be implemented in current commercial processors.

Finally, the filtering mechanism in the SMT-baseline discards a number of LQ snoops when stores write to the L1 [21]; however, it suffers from several issues, discussed in Section 2.2.1. Despite our idealized implementation of the filtering approach, these issues compromise its performance in synchronization-intensive workloads, as our experimental results show, where LQ snoops are more frequent, and results in a *worse performance than the SMT-baseline in half of*

Table 2: System Configuration.

Processor (Ice Lake like)	
Fetch width	5 instructions
Issue width	10 uops
Reorder buffer	352 entries
LQ	128 entries
SQ/SB	72 entries
Branch predictor	L-TAGE [30]
Memory dep. predictor	Store-set [15]
Memory hierarchy	
Private L1 Instruction and Data caches	L1I: 32KB, 8 ways, 4 cycles. L1D: 48KB, 12 ways, 5 cycles. Pipelined. Stride prefetcher [31].
Private L2 cache	512KB, 16 ways, 12 cycles
Shared L3 cache (16 banks)	1MB per bank, 8 ways, 35 cycles
Memory access time	160 cycles

the workloads. In addition, it also needs to store and update the LQ directory information in the L1D cache blocks.

4. Experimental Setup

We evaluate our proposal using a detailed in-house out-of-order core model, which faithfully models simultaneous multithreading. We use Sniper [27] to feed our processor model with the instructions to execute. We model the memory hierarchy, including the cache coherence protocol, using the cycle-accurate GEMS simulator [28], and the interconnect with GARNET [29].

We simulate a multicore processor consisting of 16 cores, with microarchitecture parameters resembling the Intel's Ice Lake microarchitecture [12]. Table 2 shows the main architectural parameters of the simulated system. We model a fully pipelined L1D cache with stride prefetching. We set the L1D latency to 5 cycles after actual cores [32]. The LQ and SQ/SB are also modeled in detail, including the searches for the speculative support for memory ordering, as described in Section 2. These searches are triggered in parallel to the L1D accesses. We modeled them pipelined and set their latency equal to that of the L1D cache. Despite Intel only implements 2-way SMT in their general-purpose processors, we evaluate up to 8-way SMT. Increasing the number of SMT threads increases contention in the shared resources and can complicate the design of structures such as the ROB or the register file [33]. However, this degree of multithreading is already supported by commercial processors such as the IBM POWER8 [34]. When SMT is enabled, the ROB, LQ and SQ-SB entries are statically partitioned among threads. In addition, a single thread is allowed to fetch, decode, rename, dispatch and commit instructions per cycle using a round robin policy.

We focus the evaluation on a suite of six fine-grain, synchronization-intensive, parallel benchmarks [35, 36]. Concurrent Queue (CQ) inserts and removes elements in a shared thread-safe queue, resembling write ahead logs widely used in databases and journaled file systems [37]. Persistent Cache (PC) – updates entries in a shared hash table –, RB-tree (RB) – inserts and removes nodes in a red-black tree –, and Array Swaps (SPS) – randomly swaps elements in an array –, are similar to implementations in NV-Heaps [6]. Finally, TATP and TPCC, execute the update location transactions of the TATP database workload [38] and the new order transaction in a TPCC database [7], respectively. We run the benchmarks using from 1 to 16 threads. The

number of operations is fixed (ranging from 0.4M in RB to 1.6M in TPCC) and it is evenly divided among all threads. We also evaluated SPLASH-3 [3] and PARSEC 3.0 [39] workloads, which are relatively synchronization poor.

5. Evaluation

To analyze the performance benefit provided by Spec-ITSLF, we consider five different setups increasing the number of threads from 1 to 16. The single-threaded multicore is a baseline multicore without any multi-threading support, which allocates each thread to a different core. This is the only multicore setup we analyze. The remaining setups are singlecore with X-way SMT support, where X equals the number of threads. In these setups, all threads are allocated to a single SMT core. We study the performance of a baseline X-way SMT core, resembling an state-of-the-art SMT implementation, and a filtering X-way SMT core implementing the LQ-directory approach [21] discussed earlier. In addition, we also study the performance two X-way SMT cores with inter-thread store-to-load forwarding support: the original ITSLF mechanism [8] and the Spec-ITSLF mechanism proposed in this work.

Figure 7 shows the performance benefit of the studied setups, compared to the X-way SMT singlecore, considering the optimal number of threads for each workload. When the normalized performance is above 1 the setup outperforms the baseline X-way SMT singlecore. For the single-threaded multicore, the optimal performance is achieved with 1 thread in RB and TATP, 2 threads in CQ, and 8 threads in PC, SPS, and TPCC. For the X-way SMT setups, the optimal performance is achieved with 2 threads in CQ, RB, and TATP, and 8 threads in PC, SPS, and TPCC. The only exception is the baseline SMT with the RB workload, in which case the highest performance is achieved with a single thread. Clearly, CQ, RB, and TATP put high pressure on synchronization since all threads synchronize frequently in few locks, which explains their poor performance scalability when increasing the number of threads. Synchronization is somehow lighter in other workloads and, particularly, in TPCC, which synchronizes threads using multiple locks. In summary, this figure highlights the maximum performance each setup provides.

As we already stated, the performance of fine-grain, synchronization-intensive parallel workloads does not successfully scale with the number of cores. Even though the performance of the single-threaded multicore setup improves with up to 8 threads in three of the workloads compared to the single-threaded singlecore setup, the benefits are on average moderate taking into account the additional cores it employs. Only for the TPCC workload, the single-threaded multicore greatly outperforms the baseline singlecore SMT setup. On the contrary, no performance benefit is achieved in RB and TATP, and the improvement is minimal in CQ.

Executing all the threads in a single SMT core allows for a more efficient communication through the L1 cache. It is worth noting, however, that these threads will share all the resources of the core, otherwise available to a single thread and, therefore, they will execute at a slower pace. In addition to that, the baseline SMT core also needs each store to trigger a search in the LQs of the other threads to prevent exposing a *load→load* ordering violation. This search increases contention in the LQ snoop port. Nevertheless, faster thread synchronization clearly outweighs the SMT performance-limiting factors and the SMT baseline outperforms the single-threaded multicore by 22%, on average, across the studied workloads. As mentioned earlier, TPCC and RB are the only exceptions. TPCC uses multiple locks (thus, threads need to synchronize through the LLC less frequently) and suffers particularly from LQ snoop port contention, resulting in very low performance for the baseline SMT setup. Conversely, RB has all threads synchronizing in the

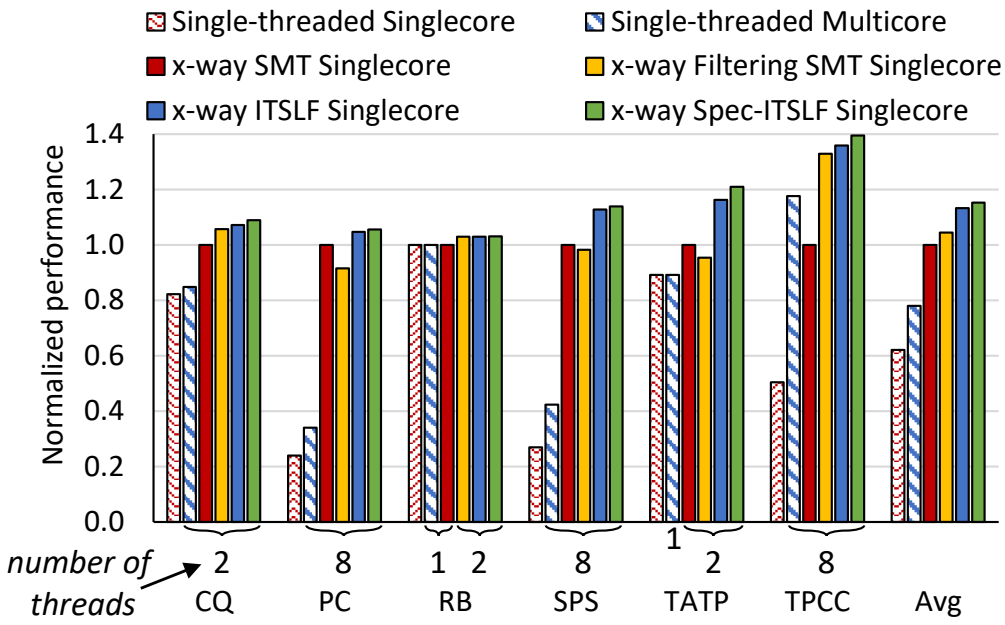


Figure 7: Performance benefit with optimal number of threads for synchronization-intensive workloads compared to the x-way SMT Singlecore setup.

same lock to execute very long critical sections, which explains its small performance sensitivity across all setups.

The filtering SMT setup significantly reduces the number of LQ snoops when stores write to memory but that comes at the cost of making store writes slower when the LQ snoop is actually required, as discussed in Section 2.2.1. Therefore, its potential benefits in synchronization-intensive workloads, where sharing cachelines is relatively frequent, are limited. In fact, compared to the baseline SMT, the filtering SMT setup only improves significantly the performance of TPCC (33%) thanks to filtering many LQ snoops, which reduces LQ snoop port contention. However, in two of the workloads (PC and TAPT) the filtering SMT setup leads to *slowdowns* compared to the SMT baseline! Despite the filtering SMT setup performs slightly better on average than the baseline SMT (5% performance benefit), the benefit *is not consistent*, which is a major disadvantage.

ITSLF brings synchronization inside the core (between the SQ/SB and the LQ), further accelerating synchronization compared to the baseline SMT core where it is done through the L1 cache. Besides, it does not require stores to snoop the LQs of other threads when writing to memory, which reduces LQ snoop port contention. Consequently, ITSLF consistently outperforms both the baseline SMT and filtering SMT, as well as, the single-threaded multicore setup, across all workloads. (The performance difference between the filtering SMT and ITSLF when running RB is minimal because, as explained before, all threads synchronize in the same lock and execute very long critical sections.) ITSLF is, on average, 13% and 8% better than the baseline SMT and filtering SMT, respectively, and outperforms the single-threaded multicore by 45%.

Finally, Spec-ITSLF further accelerates synchronization by allowing speculative stores to

forward their data to loads from other threads earlier, as soon as the stores compute their address. These cycles, which elapse from the time the store computes its address until it becomes non-speculative, are sometimes the difference between forwarding the data to load or squashing such a load. Remind that in the original ITSLF mechanism, when a load searches the SQ/SB, it cannot get the data forwarded from a different thread store if the store is still speculative. In that case, the load will get the data forwarded from an *older* store in the SQ/SB (if that store exists) or will read the data from the L1 cache. However, the load will either commit with a value that is about to be written by the other thread store (e.g., it could read that a lock is locked when another thread is about to free it) or will be squashed if the store becomes non-speculative before the load commits. Spec-ITSLF avoids these scenarios by allowing a load to get the data forwarded from a different thread store in the SQ/SB as soon as the store computes its address. Spec-ITSLF slightly outperforms ITSLF across all workloads. The performance benefit of Spec-ITSLF is on average 2% and up to 5% for the TATP workload. Spec-ITSLF advantages, however, are not limited to this performance difference but also lie on its lower implementation overhead and complexity on top of an SMT core.

Spec-ITSLF benefits are reflected in the acquire time of contended locks. We define a contended lock as a lock where there is at least one thread spinning to acquire it when it is released. Figure 8 shows the number of cycles elapsed from when a contended lock is released to the time it is acquired when running the workloads with their optimal number of threads. Threads running in a baseline SMT core require 48 cycles on average to lock a contended lock after it is released. The filtering baseline increases the average lock acquire time to 64 cycles because cachelines containing the synchronization data frequently require stores to snoop the LQ when writing to the L1, doubling the L1D access latency. ITSLF greatly reduces the lock-acquire latency, which drops to only 37 cycles, because it directly communicates the lock values within the core and merges both LQ snoops into a single one. Thanks to allowing inter-thread forwarding from speculative stores in the SQ and to trigger the LQ snoop earlier than ITSLF, Spec-ITSLF achieves an even shorter lock-acquire latency of only 34 cycles on average. Lock-acquire cycles are on the critical path of each thread and most of the cycles saved directly contribute to reducing workload execution time. Note, however, that how they impact performance of a workload also depends on the length of the critical section.

As in a single-threaded core, where we assume that speculatively executed stores will not be squashed frequently and allow them to forward their data to same thread younger loads, Spec-ITSLF allows inter-thread store-to-load forwarding from speculative stores. The ITF table keeps track of the inter-thread forwardings and guarantees that squashed stores propagate the squash to all threads when needed. Obviously, a high ratio of squashes in stores that forward to loads from other threads would undermine the benefit of Spec-ITSLF. Figure 9a presents the number of inter-thread forwarding stores that are committed and squashed per 1K committed stores. This figure clearly shows that stores that forward to loads from other threads in the core are rarely squashed. This low ratio of squashed stores is key for the performance benefit that Spec-ITSLF achieves.

Finally, Figure 9b shows the number of loads per lock acquire that read the data from the SQ/SB of another thread. Thanks to allowing inter-thread forwarding from speculative stores in the SQ, Spec-ITSLF greatly improves this metric compared to ITSLF. The improvement is particularly high in the workloads that are extremely synchronization-intensive (CQ, RB, and TATP), where there is always a load locking the lock that gets the lock-free information through inter-thread forwarding with Spec-ITSLF. The probability of inter-thread forwarding with ITSLF is lower because the inter-thread forwarding window is shorter (starts when a store becomes

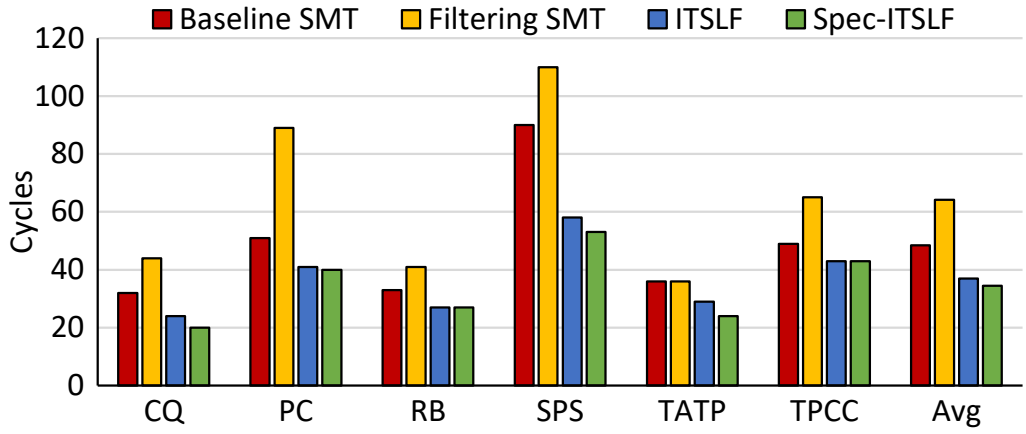
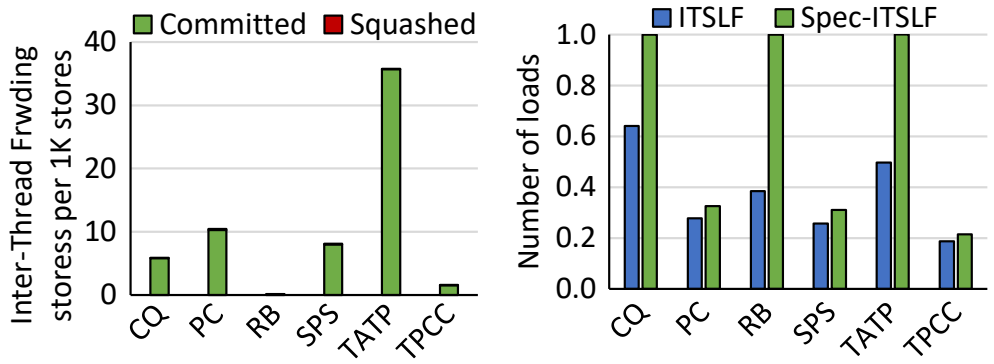


Figure 8: Average lock acquire time after lock release for contended locks.



(a) Inter-thread forwarding stores per 1k committed stores.

(b) Inter-thread forwarded loads per lock acquire

Figure 9: Inter-thread forwarding stores committed and squashed (Fig. 9a) and forwarded loads per lock acquire (Fig. 9b).

non-speculative), ranging approximately from 0.4 to 0.6 loads per lock acquire. As expected, the difference is smaller in *less* synchronization-intensive workloads (PC, SPS, and TPCC) since more often locks are released without any thread waiting to acquire them.

Synchronization-poor workloads. In addition to the synchronization-intensive parallel workloads we have discussed, we also evaluated the performance impact of Spec-ITSLF in the SPLASH-3 and PARSEC 3.0 workloads, which are relatively synchronization-poor. As we already observed in ITSLF [8], the performance benefit of Spec-ITSLF in this scenario compared to a baseline SMT core comes from avoiding the LQ snoop that stores perform when they write to memory in the baseline SMT core. Because synchronization is infrequent, accelerating the lock release – acquire operation has a very minor impact on performance. Consequently, the performance difference between Spec-ITSLF and ITSLF in this scenario is negligible. We refer the readers interested in the impact of inter-thread forwarding on performance to the experimental evaluation carried out in the original ITSLF proposal [8].

5.1. Performance sensitivity to ITF table size

The release and subsequent acquire of a lock are the critical steps to accelerate synchronization. In contended locks, when a thread is about to free a lock it holds, there is at least one thread (possibly multiple) waiting for the lock to be freed. Spec-ITSLF accelerates the communication of the lock free value (store) to the threads waiting to acquire the lock (loads) thus making synchronization faster. Because a single entry in the ITF table can save the inter-thread forwarding from a store to the loads from all threads in the core, only one ITF entry is needed to accelerate each lock release – acquire operation. After a lock is released, threads try to acquire the lock writing on it (e.g., with an atomic RMW instruction). Whether lock acquire writes are communicated faster to threads that check if the lock is free or not does not impact on performance because these threads are already too late to acquire the lock. This makes a single entry in the ITF table enough to accelerate a lock release – acquire operation.

In our workloads, we rarely observed multiple concurrent release – acquire operations to different locks, which would require several ITF entries. When moving from one to two entries in the ITF table, the maximum performance benefit is achieved by TPCC and it is only by 0.3%. Adding more ITF entries does not have a noticeable impact on performance. Therefore, we decided to implement the ITF table with only two entries and report the performance and hardware overhead of Spec-ITSLF assuming a 2-entry ITF table. Nevertheless, because each ITF entry only requires 9 bytes of storage, it is possible to implement a bigger ITF table and make Spec-ITSLF more general and able to accelerate other synchronization-intensive workloads or lock algorithms with potentially greater inter-thread forwarding *requirements*. For instance, a design choice could be to add one entry per SMT thread in the ITF table. Assuming an 8-way SMT core, the hardware overhead of this approach is 244 bytes, only 44 bytes more than the original ITSLF mechanism.

6. Potential for Long-Term Impact

Despite many parallel workloads are optimized to be as scalable as possible and minimize synchronization and communication among threads, reducing the synchronization and communication in other workloads is challenging and does not easily lead to synchronization-poor algorithms. Workloads such as those that implement graph and tree algorithms [6] or write-intensive transaction processing [7, 38] do not have straightforward synchronization-free versions because threads frequently operate with shared data and thus, require synchronization or communication with relatively high frequency. For instance, the TPCC [7] and TATP [38] workloads perform different transactions in databases and require synchronization to guarantee that the database updates are consistent and seen by all other threads before they perform their updates. Despite Spec-ITSLF achieves higher performance benefits compared to a baseline SMT when synchronization and communication are more frequent (it also reduces the LQ search port contention, which improves performance compared to a baseline SMT in several synchronization-poor workloads, as shown in [8]), we would like to emphasize that it also opens a more efficient communication channel among threads running in an SMT core, which could be leveraged by hardware and software optimizations.

Allowing direct store-to-load forwarding among threads sharing the same physical core, without first writing the stores in memory for any other external thread to see, has never been attempted. The reason is that, on the face of it, it would violate store atomicity: some threads (those sharing the physical core), would see stores earlier than other threads (those outside),

and possibly in an order different than the order the stores would be written to memory (global memory order). This would spell disaster for any memory consistency model that relies on store atomicity for correctness, which includes virtually all memory models currently in use.

However, it is exactly this that unlocks the fastest synchronization and communication that can take place between threads. Store-to-load forwarding leverages the fundamental mechanism of any core to safeguard sequential program semantics: loads look in the store buffer to find the latest store value, and stores look into the load queue to find loads that may have mistakenly taken an older value. We exploit this mechanism but now across threads. We develop techniques to guarantee coherence, store serialization, and store atomicity, while at the same time reducing hardware costs and improving performance.

The key contribution of our work is that we unlock the fastest communication level among threads which exists in the store buffer and the load queue of an SMT core. As we demonstrate in the evaluation, this impacts a class of algorithms that are notoriously difficult to scale and are becoming increasingly important in recent application trends. Taking this further, our solutions can inspire similar approaches for GPUs, accelerators, manycores, etc., where there is need for fine-grain synchronization among threads but the burden of exposing every store to the memory system for the sake of the memory model would be a considerable burden.

Looking forward, inter-thread store-to-load forwarding opens a set of challenges and possibilities that could be explored in future work and have long-term impact in future designs.

- **Heterogeneous processors.** The processor design trend is shifting from homogeneous to heterogeneous multicores, additionally including hardware accelerators to improve the execution of important workloads. Following this design trend, a super-SMT core supporting a high number of threads (e.g., 8/16 threads) with Spec-ITSLF can act as an accelerator for synchronization-intensive workloads and resort to a high-performance SMT (e.g., 2/4 threads) or to single-threaded core, depending on the workload requirements.

- **Atomic instructions.** Spec-ITSLF allows an atomic RMW instruction to forward the data to other SMT threads within the SMT core but do not allow it to read the data from the SB. This is a conservative design choice taken after single-threaded cores, where an atomic instruction always waits until it is the oldest instruction in the ROB and the SB is empty to execute. To the best of our knowledge, this constraint is kept in state-of-the-art SMT cores. Removing this constraint and allowing atomic instructions to read the data from the SB of a co-running thread can further accelerate the execution of parallel applications that rely only on atomic instructions to communicate data.

- **Software optimizations.** Workloads can also be optimized to take advantage of Spec-ITSLF. For instance, operations can be organized to favor communications among the threads running in the same SMT core, where synchronization is as cheap as common a store-to-load forwarding. Similarly, naturally imbalanced threads running in a SMT core could synchronize more frequently to minimize thread imbalance and improve performance.

- **Strong memory model with inter-thread forwarding.** Systems like the IBM z/Series provide a memory model stronger than TSO, known as IBM 370. Current SMT implementation of these systems not only disable store-to-load forwarding, but also stall loads in a matching store in another thread in executing [20]. Spec-ITSLFs, with modifications to support a stricter IBM 370 memory model [23] can significantly boost the performance of these SMT implementations.

7. Conclusion

Despite the research effort devoted to make synchronization faster and more efficient, the performance and scalability of fine-grain, synchronization-intensive, parallel workloads is strongly constrained by the cost of synchronization and communication among threads. Because communicate must be performed through a common coherent level of the memory hierarchy, increasing the number of threads usually makes communication more expensive, further exacerbating the scalability issue.

This makes SMT an attractive choice to run these workloads since threads running in an SMT core can communicate through the L1 cache. Furthermore, in our previous ITSLF proposal, we showed that the store-to-load forwarding concept can be extended to allow communication of SMT threads via their LQs and SQs/SBs while respecting rMCA.

In this work, we presented Spec-ITSLF, which extends the original mechanism to allow inter-thread forwarding from speculative stores. This allows Spec-ITSLF to further accelerate synchronization by forwarding store values earlier to other threads. When running fine-grain, synchronization-intensive parallel workloads, Spec-ITSLF outperforms a baseline SMT core by 15% on average (up to 39% for TPCC) and the original ITSLF by 2% on average (up to 5% for TATP). More importantly, Spec-ITSLF storage overhead is on par with the original ITSLF and neither requires delaying the LQ search that stores trigger to their own LQ when a store executes until the store becomes non-speculative nor needs to keep track of the speculative state of stores, which were important sources of overhead and complexity in the original proposal.

8. Acknowledgments

This work was supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant RTI2018-098156-B-C53, the European Research Council (ERC) under the Horizon 2020 research and innovation program (grant agreement No 819134), the Vetenskapsradet project 2018-05254, and the European joint Effort toward a Highly Productive Programming Environment for Heterogeneous Exascale Computing (EPEEC) (grant No 801051). Josué Feliu is supported by a Juan de la Cierva Formación Contract (FJC2018-036021-I).

References

[1] M. L. Scott, Shared-Memory Synchronization, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2013.

[2] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, in: 22nd Int'l Symp. on Computer Architecture (ISCA), 1995, pp. 24–36.

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

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

[5] N. Barrow-Williams, C. Fensch, S. Moore, A communication characterisation of Splash-2 and Parsec, in: Int'l Symp. on Workload Characterization (IISWC), 2009, pp. 86–97.

[6] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, S. Swanson, Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories, in: 16th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS), 2011, pp. 105–118.

[7] Transaction Processing Performance Council (TPC). TPC Benchmark B, `https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf` (2010).

[8] J. Feliu, A. Ros, M. E. Acacio, S. Kaxiras, ITSLF: Inter-Thread Store-to-Load Forwarding in Simultaneous Multithreading, in: 54th Int'l Symp. on Microarchitecture (MICRO), 2021, pp. 1296–1308.

[9] M. Dixon, P. Hammarlund, S. Jourdan, R. Singhal, The next-generation Intel core microarchitecture, Intel Technology Journal 14 (3) (2010) 8–28.
[10] Intel, Intel® 64 and ia-32 architectures optimization reference manual, www.intel.com (Jun. 2016).
[11] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, A Primer on Memory Consistency and Cache Coherence, Second Edition, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2020.
[12] I. E. Papazian, New 3rd gen Intel® Xeon® Scalable processor (Codename: Ice Lake-SP), in: 32nd HotChips Symp., 2020, pp. 1–22.
[13] A. Frumusanu, Apple announces the Apple silicon M1: Ditching x86 - what to expect, based on A14, https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2 (Nov. 2020).
[14] A. Fog, The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers, https://www.agner.org/optimize/microarchitecture.pdf (Mar. 2021).
[15] G. Z. Chrysos, J. S. Emer, Memory dependence prediction using store sets, in: 25th Int'l Symp. on Computer Architecture (ISCA), 1998, pp. 142–153.
[16] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, 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), 2011, pp. 694–701.
[17] M. Dubois, M. Annavaram, P. Stenström, Parallel Computer Organization and Design, Cambridge University Press, 2012.
[18] K. Gharachorloo, A. Gupta, J. Hennessy, Two techniques to enhance the performance of memory consistency models, in: 20th Int'l Conf. on Parallel Processing (ICPP), 1991, pp. 355–364.
[19] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, M. O. Myreen, x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors, Communications of the ACM 53 (7) (2010) 89–97.
[20] K. J. Alexander, C. J. Jonathan T. Hsieh, M. Recktenwald, Load and store ordering for a strongly ordered simultaneous multithreading core, U.S. Patent US14511408 (Oct. 2014).
[21] A. D. Hilton, A. Roth, SMT-directory: Efficient load-load ordering for SMT, IEEE Computer Architecture Letters 9 (1) (2010) 25–28.
[22] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, P. Sewell, Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8, in: 45th Symp. on Principles of Programming Languages (POPL), 2018, pp. 19:1–19:29.
[23] A. Ros, S. Kaxiras, Speculative enforcement of store atomicity, in: 53rd Int'l Symp. on Microarchitecture (MICRO), 2020, pp. 555–567.
[24] H.-J. Boehm, S. V. Adve, Foundations of the C++ concurrency memory model, in: 2008 Conf. on Programming Language Design and Implementation (PLDI), 2008, pp. 68–78.
[25] A. Buyuktosunoglu, A. El-Moursy, D. H. Albonesi, An oldest-first selection logic implementation for non-compacting issue queues, in: 15th Annual Int'l ASIC/SOC Conference, 2002, pp. 31–35.
[26] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, M. Själander, Efficient invisible speculative execution through selective delay and value prediction, in: 46th Int'l Symp. on Computer Architecture (ISCA), 2019, pp. 723–735.
[27] T. E. Carlson, W. Heirman, L. Eeckhout, Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations, in: Conf. on Supercomputing (SC), 2011, pp. 52:1–52:12.
[28] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, D. A. Wood, Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset, ACM SIGARCH Computer Architecture News 33 (4) (2005) 92–99.
[29] N. Agarwal, T. Krishna, L.-S. Peh, 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), 2009, pp. 33–42.
[30] A. Seznec, The L-TAGE branch predictor, The Journal of Instruction-Level Parallelism 9 (2007) 1–13.
[31] J.-L. Baer, Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors, 1st Edition, Cambridge University Press, 2009.
[32] S. Shukla, S. Bandishte, J. Gaur, S. Subramoney, Register file prefetching, in: 49th Int'l Symp. on Computer Architecture (ISCA), 2022, pp. 410–423.
[33] M. Nemirovsky, D. M. Tullsen, Multithreading Architecture, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2013.
[34] B. Sinharoy, J. A. V. Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, K. M. Fernsler, IBM POWER8 processor core microarchitecture, IBM Journal of Research and Development 59 (1) (2015) 2:1–2:21.
[35] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, T. F. Wenisch, Language-level persistency, in: 44th Int'l Symp. on Computer Architecture (ISCA), 2017, pp. 481–493.
[36] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, T. F. Wenisch, Persistency for synchronization-free regions, in: 39th Conf. on Programming Language Design and Implementation (PLDI), 2018, pp. 46–61.

[37] S. Pelley, P. M. Chen, T. F. Wenisch, Memory persistency, in: 41st Int'l Symp. on Computer Architecture (ISCA), 2014, pp. 265–276.
[38] S. Neuvonen, A. Wolski, M. Manner, V. Raatikka, Telecom application transaction processing benchmark, `http://tatpbenchmark.sourceforge.net/` (2011).
[39] C. Bienia, Benchmarking modern multiprocessors, Ph.D. thesis, Princeton University (Jan. 2011).