ITSLF: Inter-Thread Store-to-Load Forwarding in Simultaneous Multithreading

ABSTRACT

In this paper, we argue that, for a class of fine-grain, synchronization-intensive, parallel workloads, it is advantageous to consolidate synchronization and communication as much as possible among the threads of simultaneous multithreading (SMT) cores. While, today, the shared L1 is the closest coherent level where synchronization and communication between SMT threads can take place, we observe that there is an even closer shared level, entirely inside a single core. This level comprises the load queues (LQ) and store buffers (SB) of the SMT threads and to the best of our knowledge it has never been used as such. The reason is that if we allow communication of different SMT threads via their LQs and SBs, i.e., ***inter-thread store-to-load forwarding (ITSLF)***, we violate write atomicity with respect to the outside world, beyond the acceptable model of read-own-write-early multi-copy atomicity (rMCA).

The key insight of our work is that we can accelerate synchronization and communication among SMT threads with *inter-thread store-to-load forwarding*, without affecting the memory model—in particular without violating write atomicity (rMCA). We demonstrate how we can achieve this entirely through speculative interactions between LQs and SBs of different threads, while ensuring deadlock-free execution. Without changing the architectural model, the ISA, or the software, and without adding extra hardware in the form of a specialized accelerator, our insight enables a new design point for a standard architecture. We demonstrate that with ITSLF, workloads scale better on a single 8-way SMT core (with the resources of a single-threaded core) than on a baseline SMT (with or without optimizations), or on 8 separate single-threaded cores.

1. INTRODUCTION

Synchronization and transfer of critical data from thread to thread has always been a centerpiece concern for the performance of shared-memory parallel workloads. A vast body of work aims to make synchronization algorithms more performant and more efficient— [37] provides a good review—but we do not expand on it here as it is orthogonal to the perspective we take in this paper. Despite these advances, synchronization tends to be avoided whenever possible due to its detrimental effects on performance and scalability. It is telling that traditional parallel benchmark suites such as SPLASH [36, 43] and PARSEC [7] are relatively synchronization-poor [5].

While many workloads similar to SPLASH [36, 43] and PARSEC [7] scale well in modern multicores, a different class of fine-grain, synchronization-intensive, parallel workloads performs poorly, progressively underutilizing core resources with more cores. For example, workloads such as those that implement graph and tree algorithms [13] or write-intensive transaction processing [40] belong to this class. Recently, this class of workloads has become increasingly relevant in many research areas, including, for example, memory persistency work, e.g., [22]. A critical reason for this is that the farthest synchronization has to reach, the more expensive it becomes. For example, synchronizing two cores via their shared last-level cache (LLC) is more expensive than synchronizing two simultaneous multithreading threads via their shared L1. This not only holds for the actual synchronization operations, but equally important, for the transfer of critical data from thread to thread (the reason why synchronization was needed in the first place).

Trying to scale fine-grain, synchronization-intensive workloads is often an exercise in frustration as the more resources (cores) we allocate to run, the more expensive thread synchronization becomes. The key reason is that synchronization in shared memory is fundamentally achieved as a data race of conflicting accesses that exposes the order between them. As such, the conflicting accesses participating in the data race must be globally inserted in the memory order to be visible by all synchronizing parties. This implies that synchronization must be achieved via a common coherent level of the memory hierarchy. Because of the hierarchical cluster structure of modern systems (core clusters inside multicores, processor packages, NUMA nodes, etc.), chances are, the more cores we use, the farther away their common coherent level will be found, making synchronization increasingly expensive.

Research Question: This opens up the main question we tackle in this paper. *How can we bring communication and synchronization even closer to the executing threads?* Specialized synchronization instructions and additional helper hardware structures have been proposed to accelerate synchronization in other settings, e.g., in processing in memory MiSAR [28] and SynCron [20]. However, our aim is to examine the problem *without changing the architectural model, or the ISA, or the software, and without adding any new hardware structures*. This opportunity is afforded by Simultaneous Multithreading (SMT) [42].

Key Insight: The key observation of our work, and one that as far as we know has not been exploited before, is that

the first shared level between threads in an SMT architecture is not the L1 cache (as it is usually done) but the *Store Buffers* (*SB*) that serve each of SMT threads. In other words, if we were allowed to forward data from the stores in the SB of one thread to the loads of another thread, we could then achieve faster and more efficient *in-core synchronization and communication*, through ***inter-thread store-to-load forwarding (ITSLF*** [1]***)***.

Why ITSLF has not been done before? There are a few reasons that prevented the closer examination of inter-thread store-to-load forwarding in the past:

- *Stores in the store buffer are not in the memory order yet*. This has significant implications for the memory model. Specifically, allowing inter-thread store-to-load forwarding violates store atomicity beyond the acceptable rMCA (read-own-write-early multiple-copy atomic) model [1, 41] (Section 3). This is also the reason why the SB is logically partitioned (irrespective of its physical organization) amongst threads for memory models that require rMCA store atomicity.

- While speculative enforcement of store atomicity, as a consequence of speculative enforcement of sequential consistency (SC), is known [8, 18, 21], it was not until recently that it was shown, in related work, that efficient implementations can hone in on the specific situations that cause store atomicity violations and effectively enforce it only when it is needed [35]. We draw inspiration from the ideas in [35] but our main contribution is to build a comprehensive approach to address a multitude of issues stemming from ITSLF, issues that are described in Section 3.

- It is not required in SMT scenarios where independent threads are running (no chance for forwarding). In fact, even for synchronization-poor parallel SMT threads running on the same core, the opportunities for forwarding are few: Consider that modern parallel software is data-race-free (DRF) and communication between threads during large synchronization-free regions is simply non-existent. The value of forwarding, however, becomes apparent in synchronization-intensive workloads such as the ones we examine here: first, for the synchronization variables themselves, but more-so for the critical section data that need to be transferred from the stores of one thread to the loads of another after a successful lock hand-off.

What are our main results?

- We demonstrate effective store-to-load forwarding from the store buffers of SMT threads, leading to significant increase in performance for a number of synchronization- and write-intensive workloads (Section 5).

- Furthermore, contrary to the prevailing view that SMT is not worth scaling beyond two or four threads per core, we show that, for these workloads, a "Super-SMT" approach of up to 8 (or potentially more) threads is beneficial for scaling performance *even with the resources of the baseline non-SMT core* (we only scale the architectural state with the number of threads but nothing else–see Sections 2 and 5). In SMT implementations, two threads with moderate ILP experience only a small speed up as they compete for the same resources. In our case, however, our target workloads are dominated by synchronization and communication latencies, which is exactly what multithreading can hide by increasing the utilization of the core's resources with more threads. This is the same phenomenon exploited by GPUs when they switch to a different thread when issuing long-latency memory operations.

Overall, we demonstrate a straightforward approach to provide increased performance and better scaling to a class of workloads that traditionally have been difficult to accelerate. We do not resort to a synchronization accelerator solution that incurs significant changes (it would require additional hardware and changes in the architectural model, ISA, and software). Instead, our approach leaves the architectural model unchanged and is orthogonal to software and synchronization optimizations. We bring synchronization and communication one level closer than the L1 to the executing threads.

What are our main contributions?

- For the first time (as far as we know) we solve the problems that arise with ITSLF in an SMT setting. In particular, we determine the point when a store becomes locally visible to SMT threads, we safeguard write serialization for same-address stores while they are only locally visible in the SMT (but not globally visible outside the SMT), and we efficiently maintain read-own-early Multi-Copy Atomicity (rMCA) both within and outside the SMT using speculation (Section 3).

- We demonstrate how our efficient implementation of ITSLF reduces the number expensive CAM searches compared to the baseline non-ITSLF SMT-baseline (Section 5).

- Finally, we show that synchronization-intensive workloads consistently benefit from ITSLF, by scaling well beyond the SMT-baseline or even its load queue search filtering optimizations that were previously proposed [23] (Section 5).

2. BACKGROUND

In a Simultaneous Multithreading core most of its resources are shared among multiple threads so they can simultaneously execute as if they were placed in independent "virtual" cores (see Figure 1).

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). This approach resembles Intel's implementation of SMT [14] 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

[1]Pronounced as *itself*, as it is the own microarchitecture–using the SB–which "sends" data from one thread to another, instead of the L1.

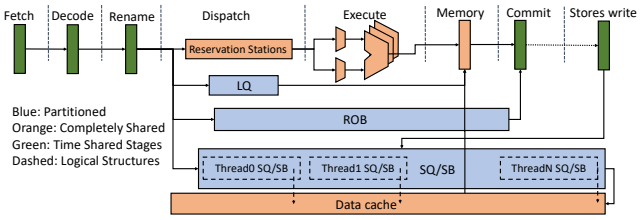


Figure 1: SMT Model.

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 [14]. 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 [24]. This implementation is orthogonal to our approach.

Atomic instructions typically empty the SB [30] when they can execute and commit. In SMT, an atomic instruction empties the SB of its own thread 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 ordering are respected. These checks require frequent associative searches on the LQ and the SQ/SB. These queues 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 [32] or Apple's M1 [17]. 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[2] 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 that five cycles in an Intel Ice Lake [16].

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 load breaks a memory dependence. This is a rare event due to accurate memory dependence prediction [12] 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 [27] 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 [32].

External stores searching the LQ: *Load→load* order is required for loads to the same address to guarantee coherence [15], and for loads to different addresses in systems that provide strong consistency guarantees [19], such as x86-TSO [38]. 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

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

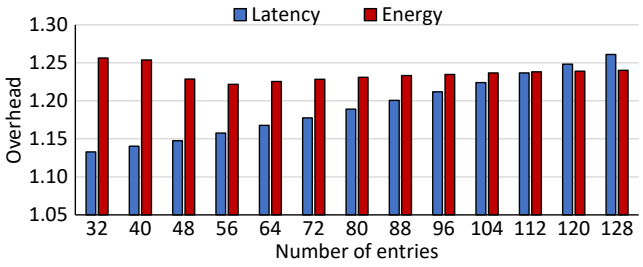


Figure 2: Search latency and energy overheads of adding a second search port to the LQ (CACTI-P).

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 to be squashed.[3] 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.

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 [23]. 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 [23]. 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, when the LQ snoop is actually required, store writes become significantly more costly. First, 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. Second, since the write cannot be performed until the LQ search is completed, a second cache access should be initiated along with the LQ snoop to perform the write. And third, since writes requiring a LQ snoop take longer to complete, the write port should be squashed to prevent store-store ordering violations within the same thread, similarly to how a L1 miss write would be managed. In addition to that, storing the LQ directory information along the L1 cachelines, also has an overhead of N bits per cacheline, with N equal to the number of supported SMT threads in the core. Because of these reasons, the LQ-directory solution is *not consistently better* than the baseline, leading in some cases to performance degradation as we show in Section 5.

3. ISSUES AND SOLUTIONS WITH ITSLF

Allowing inter-thread store-to-load forwarding from a thread to another in the same core in a 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.

However, as stated by Nagarajan et al. [30], 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 store atomicity which is respected by most vendors (e.g., x86-TSO [38], ARMv8 [34]), resulting in a more complex non-MCA model where stores are not globally ordered. To the best of our knowledge, this is the first discussion in the literature about the coherence/consistency impact of inter-thread forwarding in SMT.

We first focus on the correctness of inter-thread forwarding within a SMT core. Then, we discuss the interaction with other cores.

3.1 Point of Local Visibility

In the single thread case, stores that resolve their address

[3]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. [3] 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.

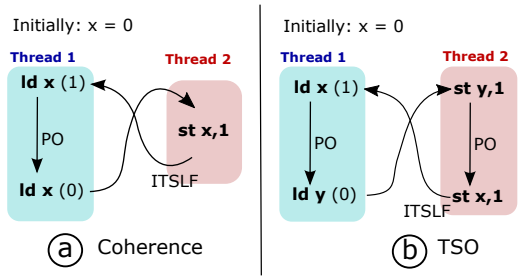


Figure 3: Coherence (a) and TSO examples (b).

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 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. [15]. Note that this example applies to every memory model. The value in between parenthesis in the loads is the value read by them. It can 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. 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.

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 SLF 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 from the point the stores compute their address, then they should search the LQs of other threads at that time. Note that, in the same thread, younger loads always *see* the thread's own stores from the time their address is available, i.e., from when the stores resolve their address in the thread's SQ. If we allow forwarding to other threads only for *committed* stores in the SB, the stores should search other LQs on commit (when they transition from the SQ to the SB). A key realization to make both local thread SLF 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.*[4] At that point the store combines its local LQ squash with the squash of other thread M-speculative loads in other LQs. We arrive, then, at the following invariant:

Invariant: *Stores become locally visible to SMT threads when they commit and pass from the SQ to the SB. When they become visible, they squash (on commit) the younger matching D-speculative loads in their own thread and any matching M-speculative load in all other threads. After a store becomes locally visible it can forward its data to loads of other threads.*

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.

An alternative is to squash earlier (as soon as the store address is available) and allow forwarding to all threads from a store that is still in the SQ, i.e., *a store that might be speculative* (e.g., from branch prediction). This leads to higher complexity and we have not found strong evidence that it offers better performance, therefore we leave it for future examination. For these reasons, and for the rest of

[4] As a thought experiment, imagine that the address of a store always becomes available when the store is at the head of the ROB.

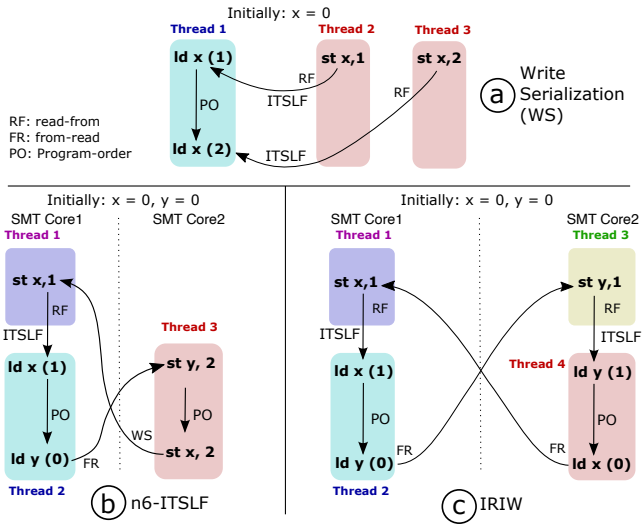


Figure 4: Write serialization (a), n6-ITSLF (b), IRIW (c).

the paper, we consider that ITSLF concerns only committed (non-speculative) stores.

Cost: 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).

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).

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 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 L1 in arbitrary order.) This means that it is impossible to decide on a local order without first knowing the global order.[5]

[5]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.

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 commit order*. When a load snoops all SBs and matches several candidates, in more than one SB, these candidates are ordered by their LV order. Right at that point the load selects the "youngest-to-LV" store for forwarding and deactivates the forwarding of all other "older-to-LV" stores.

Now combine this approach with the Invariant of the Point of Local Visibility, discussed above: Whenever a store commits and enters the SB (from the SQ) it squashes all speculative loads on the same address that may have forwarded from older-to-LV stores. This store becomes the youngest-to-LV and prevents all older-to-LV stores from ever forwarding again while they are in the SB.

At this point you may be concerned that the LV order of the stores may be different from their 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 store atomicity 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.

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

Cost: Similar functionality already exists in the ST-baseline unified FIFO 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 SMT case, a load can match multiple stores in multiple SBs (that all add up to the ST-baseline SB). We select the younger-to-commit and, in parallel, deactivate forwarding for all the ones that were not selected. We extend the SQ entries with a field to store their commit order. This field requires $\lceil \log_2(\text{SB entries}) \rceil + 1$ (sorting-bit [10] to handle wrap-around) bits. In an Ice Lake core with a 72-entry store buffer, this accounts for 8 bits per SQ entry (576 bits in total for the SQ).

3.3 Store Atomicity

Finally, we address the main culprit that prevents ITSLF in current systems: violation of store atomicity. Informally, in memory models that demand store atomicity, 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 SBs. Obviously, this would violate store atomicity in SC [26], x86-TSO [38], or even in relaxed memory models such as ARMv8 [34].

In recent work, Ros and Kaxiras [35] show that a detection of a store atomicity 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 SLF, is incorporated in memory model definitions such as x86-TSO [38] and is known as *read-own-write-early multiple-copy atomicity (MCA)*. 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 [38], Figure 4(b)) and IRIW [9] (Figure 4(c))—discussed below—are not due to read-on-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 SLF 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, read-own-write-MCA 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.

ITSLF Solution.

When a load gets the data forwarded from a store, it records the position ($\log_2$(SB entries)) of the forwarding store in the SB. Note than in a partitioned SB, the position also indicates the hardware thread the entry belongs to. A load at the head of the ROB, checks if the entry of the store still contains the forwarding store, or otherwise, the store has been written to L1 and the entry is freed. In the first case, the load will not be committed. To know if the store is still in the SB, we leverage the concept of the sorting-bit proposed by Buyuktosunoglu et al. [10]. You can think of using a sorting-bit as being equivalent of having a monotonically-increasing numbering for stores. The technique is explained in [10]. The sorting-bit augments the store's position in the SB and it is sent to the load on forwarding. If the bit stored by the load matches the one of the SB entry, then the store is still present in the SB. Checking on one single bit lets the load decide if can commit or not.

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.

Cost: ITSLF entails negligible storage overhead. We extend each LQ entry with two fields: i) a single-bit field to indicate if the load was forwarded from a different thread, and ii) a field to store the augmented position of the forwarding store. The latter only needs $\lceil \log_2(\text{SB entries}) \rceil + 1$ (sorting-bit) bits. In an Ice Lake core with a 72-entry store buffer, 8 bits per LQ entry are needed (1024 bits in total for the LQ). Overall, the storage overhead of ITSLF on Ice Lake is 1600 bits (200 bytes).

3.4 Summary

Table 1 summarizes the main actions performed by the ST-baseline, SMT-baseline, and 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 in the SQ/SB and other-thread stores in the SB, which makes synchronization communication among threads faster. Second, ITSLF gets rid of the LQ search when stores are performed and write to memory. This reduces LQ snoop port contention and helps improve the SMT performance when running synchronization-poor workloads. While the filtering mechanism in the SMT-baseline discards a number of LQ snoops when stores write to the L1 [23], it suffers from an important area overhead in the L1 (where LQ-directory information, used to filter LQ snoops, is stored) and longer latency when the LQ snoop should be triggered after searching the LQ directory. The latter compromises 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 the workloads*.

4. EXPERIMENTAL SETUP

We evaluate our proposal using a detailed in-house out-of-order core model, which faithfully models simultaneous multithreading. The core model is driven by a Sniper [11] front-end. We model the memory hierarchy, including the cache coherence protocol, using the cycle-accurate GEMS simulator [29], and the interconnect with GARNET [2].

We simulate a multicore processor consisting of 16 cores, with microarchitecture parameters resembling the Intel's Ice Lake microarchitecture [32]. Table 2 shows the main architectural parameters of the simulated system. 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 [22,25]. 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 [33]. 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

Table 1: Summary of the actions performed by the ST-baseline, SMT-baseline, and ITSLF for load & store execution.

	ST-baseline	SMT-baseline	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 the L1 when other thread is about to free it.***	**Search SQ/SB.** Forward data from most-recent ST in own-thread SQ or from unique (youngest-to-local-visibility) ST in SB (all threads). ***Read other thread freeing the lock directly from the SB!***
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 retire/commit for non-speculative forwarding)
ST retire	–	–	**Search (all threads) LQ.** Squash matching speculative LDs from any thread (only younger from own thread).
ST performed	**Write L1.**	**Write L1. Search (all threads, except own) LQ.** **(Filtering: Search LQ only if other threads share the cacheline.)** Squash matching LDs from other threads.	**Write L1. Release any waiting forwarded loads.** If forwarded to load(s): release load(s) waiting to retire by writing to the commit-gate register of the corresponding thread(s).

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 [39]
Memory dep. predictor	Store-set [12]
Memory hierarchy	
Private L1 Instruction and Data caches	Instruction: 32KB, 8 ways, 4 hit cycles. Data: 48KB, 12 ways, 5 hit cycles. Both pipelined and with a stride prefetcher [4].
Private L2 cache	512KB, 16 ways, 12 hit cycles
Shared L3 cache (16 banks)	1MB per bank, 8 ways, 35 hit cycles
Memory access time	160 cycles

similar to implementations in NV-Heaps [13]. Finally, TATP and TPCC, execute the update location transactions of the TATP database workload [31] and the new order transaction in a TPCC database [40], 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. In addition, we present results for the SPLASH-3 [36] and PARSEC 3.0 [6] workloads, which are relatively synchronization poor.

5. EVALUATION

5.1 Performance impact of ITSLF in synchronization-intensive workloads

Figure 5 shows how performance varies when increasing the number of threads from one to sixteen in four different setups. The first setup, labelled as "ST" in the figure, allocates each thread to a different core in a multicore. The other three setups allocate all threads to a single X-way SMT core, where X equals the number of threads. The second setup refers to a baseline SMT core. The third setup, "Filtering SMT", uses an LQ-directory with each cacheline to filter LQ snoops as described in [23] (see Section 2.2). Finally, the fourth setup, ITSLF is our proposal, a SMT core supporting inter-thread store-to-load forwarding. Performance is normalized to the single-thread execution. Hence, a value above 1 for the normalized performance means that the multi-threaded execution outperforms the single-thread execution.

As we already anticipated, trying to improve the performance of fine-grain, synchronization-intensive workloads by increasing the number of cores is pointless. The performance of some of these workloads directly drops when increasing the number of threads if each thread is allocated to a different core compared to running the workload with a single thread (e.g., TATP and CQ). Synchronization is somehow lighter in other workloads such as PC or SPS, which allows scarce performance benefits when increasing the core count (1.64× for SPS and 1.50× for PC). This is not productive taking into account the additional resources (and power consumption) each core adds. TPCC is the workload whose performance scales better with the number of cores (2.43×) since it uses multiple locks and thus threads need to synchronize through the LLC less frequently. Conversely, RB has all threads synchronizing in the same lock and threads spend most of their time executing long critical sections, which explains its performance insensitivity when increasing the core count.

Executing all the threads in a single SMT core allows for a more efficient synchronization 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 lower 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. Faster thread synchronization clearly outweighs the SMT performance-limiting factors. For instance, the normalized performance of PC and SPS grows up to 4.09× and 3.84×, respectively, compared to the single-thread execution. Only in RB and TPCC the baseline SMT performs worse than allocating each thread to a different core. As discussed before, these workloads benefit less from a faster synchronization and, in this case, the performance-limiting factors of SMT execution kill the benefits. The filtering SMT baseline 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

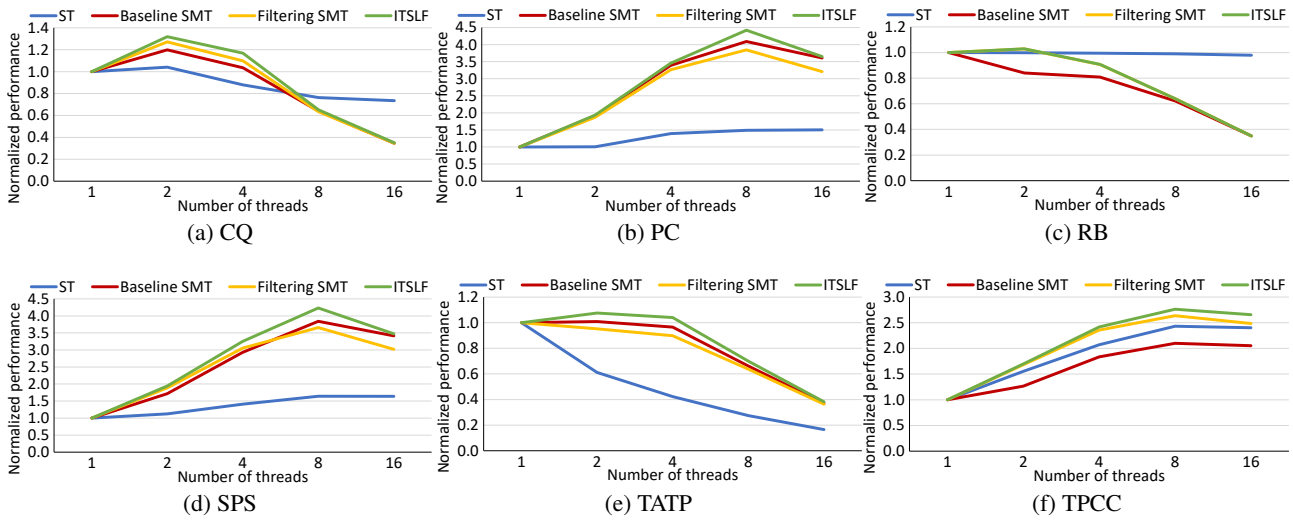


Figure 5: Performance normalized to single-thread execution times when increasing the number of threads in the multicore ST, baseline SMT, filtering SMT, and ITSLF setups.

LQ snoop is actually required, as discussed in Section 2.2. Therefore, its potential benefits in synchronization-intensive workloads, where sharing cachelines is relatively frequent, are limited. In fact, compared to the baseline SMT, it only improves significantly the performance of TPCC (from 2.10× for the baseline SMT to 2.64× for the filtering SMT). In half of the workloads (PC, SPS, TAPT) the filtering SMT baseline can lead to *slowdowns* compared to the SMT baseline! The benefit of the the filtering baseline, therefore, *is not consistent*, which is a major disadvantage.

ITSLF brings synchronization inside the core (between the 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 outweighs all other setups. More importantly, it outperforms the multicore setup, where each thread is allocated to a different core in all workloads, unlike the baseline and filtering SMT cores.

Independent of the interest to observe how performance varies when increasing the thread count, a system administrator aims to run each workload with its optimal number of threads. That is, two threads for CQ, RB, and TATP, and eight threads for PC, SPS, and TPCC. From now on, we will report results for the optimal number of threads for each workload. Figure 6 shows the performance benefit, compared to the single-thread execution, achieved by the four studied setups, considering the optimal number of threads for each workload. Therefore, the figure highlights the maximum performance each setup provides. On average, executing each thread on a different core improves performance by 1.4× compared to the single-thread execution. The baseline SMT and filtering SMT setups perform similarly (2.21× and 2.24×, respectively, compared to the single-thread execution). ITSLF outperforms the single-thread execution by 2.51×, the single-threaded multicore execution by 1.74×,

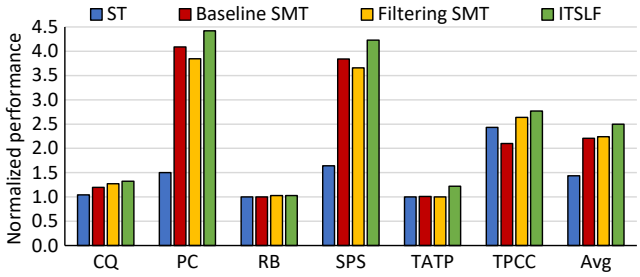


Figure 6: Performance benefit with optimal number of threads for synchronization-intensive workloads.

and the baseline SMT and filtering SMT setups by 14% and 11%, respectively.

5.2 Where does performance come from?

ITSLF accelerates synchronization.

The more obvious way ITSLF accelerates synchronization-intensive workloads is by making the transfer of critical synchronization data from thread to thread faster. This makes the data values visible significantly earlier than in the baseline SMT core, where they are communicated through the L1 cache. Note that to make a value visible to other threads in the baseline SMT processor, the store should be the oldest store of the thread in the SB and it should be the turn of the thread to write to the L1. The situation can be worse with the filtering baseline, since it should perform an LQ snoop *after* it determines if it is required. If a snoop is required (which is common in the synchornization-intensive workloads), the propagation of the write must be delayed until the snoop is completed. Thus, write latency doubles, since the write can only be performed after the LQ is snooped. This is in sharp contrast with ITSLF, which makes the data value available

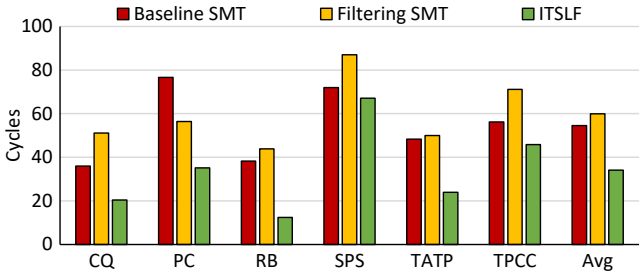
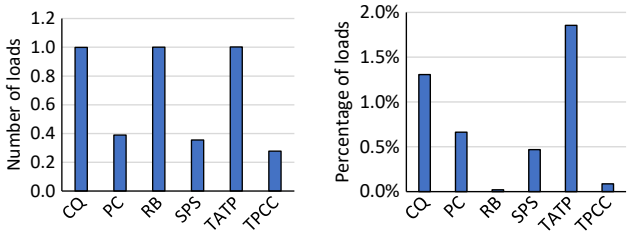


Figure 7: Average lock acquire time after lock release for contented locks.



(a) Number of loads per lock acquire

(b) Percentage of loads

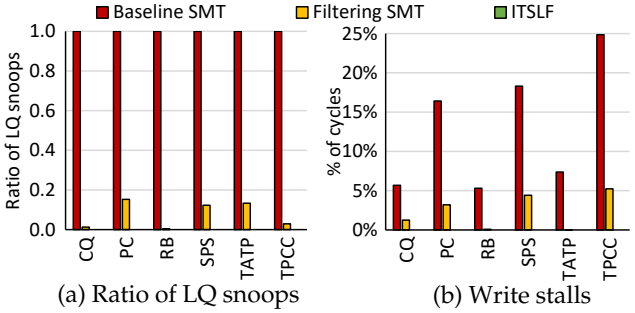
Figure 8: Loads reading a value from the SB of a different thread.

for the other threads in the core as soon as the store commits.

To quantify how much ITSLF accelerates lock acquires, Figure 7 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. We define a contended lock as a lock where there is at least one thread spinning to acquire it when it is released. Compared to the baseline SMT, the filtering baseline increases the average lock acquire time. This is caused by the cachelines containing the synchronization data, which frequently require stores to snoop the LQ when writing to the L1. On average, the lock-acquire latency grows from 55 cycles in the baseline SMT core to 60 cycles in the filtering SMT core. Thanks to directly communicating the lock values within the core, ITSLF greatly reduces the lock-acquire latency, which drops to only 34 cycles. 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.

ITSLF accelerates synchronization because threads spinning on a lock execute loads that read the synchronization data from the SB of the thread releasing the lock. Figure 8a shows the number of loads per lock acquire that read the data from the SB of another thread. CQ, RB, and TATP are extremely synchronization-intensive and no performance benefit is reached with more than two threads. This is reflected in these results: there is always a thread spinning on the lock when it is released and reads the value from the SB of the thread releasing the lock. PC, SPS, and TPCC are *less* synchronization-intensive because threads synchronize with multiple locks. That explains their performance scalability up to eight threads, which is also reflected in these results. The number of threads that read the synchronization data from the SB of a different thread in their case ranges from 0.3× to 0.4×, per lock acquire, which means that, often, locks are released without any thread waiting to acquire them.

Finally, Figure 8b shows the percentage of loads that receive the data through forwarding from a different thread with ITSLF. Obviously, the percentage is low since only loads involved in synchronization can actually read data through another thread SB. In data-race-free software, communication between threads during large synchronization-free regions is non-existent.



(a) Ratio of LQ snoops

(b) Write stalls

Figure 9: Ratio of LQ snoops per store and Percentage of cycles where a write is stalled due to LQ port contention.

ITSLF reduces contention in the LQ snoop port.

In the SMT baseline, as discussed in Section 2.2, all stores search the LQ twice: when they execute and when they write to memory, which affects performance and increases energy consumption. The filtering SMT baseline tackles this problem using the LQ-directory, and only triggers LQ snoops when required. Unlike these approaches, ITSLF does not need to snoop the LQ of any threads when stores write to memory.

Figure 9(a) shows the ratio of LQ snoops when stores write for the three SMT setups. While all stores snoop the LQ when they write with the non-filtering baseline, as the figure shows, the filtering SMT baseline reduces the ratio of snoops per store to between 1% (RB) and 16% (PC). An interesting observation that shows how the additional LQ snoops impacts performance is that in the three workloads where the filtering SMT avoids almost all LQ snoops, it outperforms the baseline SMT. In contrast, when the LQ snoops exceed 10% of the stores, the filtering SMT performs *worse* than the baseline SMT. As discussed, ITSLF does not trigger any LQ snoop when stores write, which clearly contributes to its superior performance.

Given that, in the SMT baseline, the LQ snoop port should be shared by stores that execute and stores writing to the L1, it is possible to find stores at both stages requiring the use of the LQ snoop port in the same cycle. When this happens, we simply stall the execution of the store by a cycle giving priority to the store that writes. Figure 9(b) shows the percentage of cycles where a thread executing a store

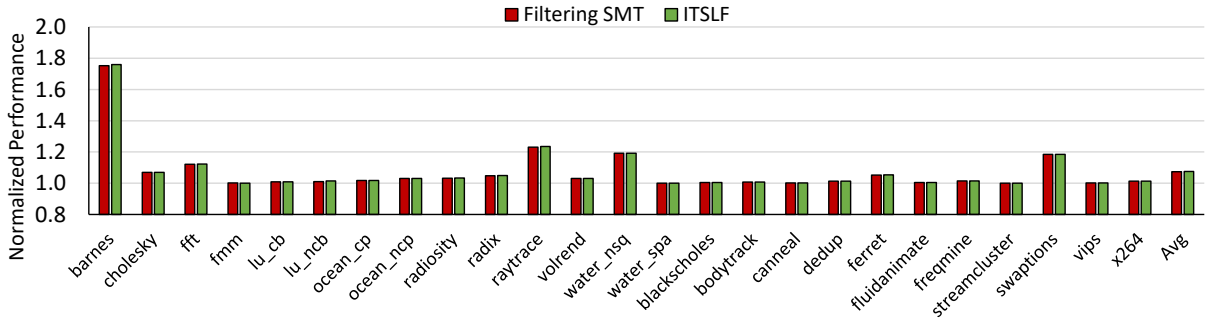


Figure 10: Normalized performance (to the baseline SMT) across SPLASH-3 and PARSEC 3.0 workloads.

suffers a stall because the LQ snoop port is used by a store writing to the L1. Note that the situation does not occur with ITSLF since no LQ search is required when stores write to memory. With the baseline SMT, three workloads (PC, SPS, and TPCC, the ones that we execute with eight threads) suffer store execution stalls due to contention in the LQ snoop port in more than 15% of the execution cycles. TPCC is the workload that suffer the highest percentage of stalls (25%) which contributes to the low relative performance that the SMT baseline achieves in this workload. The filtering SMT reduces the LQ snoops required when stores write and thus, reduces the number of stalls, which fall below 5% of the cycles in all workloads.

5.3 Performance impact of ITSLF in synchronization-poor workloads

Now, we evaluate other parallel workloads that are relatively synchronization-poor. Therefore, the faster synchronization provided by ITSLF marginally translates into performance benefits. However, ITSLF also avoids the LQ snoops that stores perform when they write to memory in the baseline SMT processor, which can have an important impact on performance in some workloads.

Figure 10 shows the performance of the filtering SMT and ITSLF compared to the baseline SMT across the SPLASH-3 and PARSEC 3.0 workloads running with eight threads. Despite this synchronization-poor scenario does not align favorably to gain performance via inter-thread store-to-load forwarding, it is very interesting to observe that avoiding the LQ snoop when stores write to memory also brings important performance benefit in many workloads. For instance, ITSLF and filtering SMT outperform the SMT baseline on barnes (where LQ snoop port contention strongly hurts performance in the SMT baseline), raytrace, water_nsquared, and swaptions by 76%. 24%, 19%, and 18%, respectively. On average, ITSLF improves performance by 8% compared to the baseline SMT core across all SPLASH and PARSEC workloads.

In this synchronization-poor scenario, the filtering SMT performs almost as good as ITSLF. Since synchronization is infrequent, the performance benefit achieved by ITSLF comes mostly from reducing the LQ snoop port contention, something that the filtering SMT baseline also achieves. However, it is important to empathize that the filtering SMT comes with an area overhead in the L1 cache since it requires storing LQ-directory information for each cacheline (one bit per SMT thread per cacheline) used to determine if the LQ of other threads should be searched. In addition, as discussed in Section 2.2, since it requires checking the LQ-directory to know if it should search the LQ of any thread, the latency of store writes is larger when the LQ snoop is finally required. This makes its performance not consistently better than the SMT baseline in synchronization-intensive workloads. The lower overhead and superior performance in all synchronization-intensive workloads clearly makes ITSLF a better approach than the filtering SMT.

6. CONCLUSION

Trying to scale fine-grain, synchronization-intensive workloads is often an exercise in frustration. The more cores we allocate to run, the farther away their common coherent level is found, making their synchronization increasingly expensive. This makes SMT an attractive choice to run these workloads: While threads running in a multicore need to synchronize through the cache hierarchy, threads running in an SMT core can do so through the L1 cache. Interestingly, even though it has never been used as such, in this work we show that there is an even closer shared level that can be used to accelerate thread's synchronization within an SMT core: the store buffer.

We propose inter-thread store-to-load forwarding (ITSLF) and address problems that arise when allowing a thread to read the data from the store buffer of another thread, earlier than when such data become globally visible. We define the point where a store becomes locally visible to other threads in the core and a visibility order for same-address stores from different threads, and show how ITSLF guarantees store atomicity using speculation, with minor changes in the architecture and negligible cost.

Our results show that ITSLF accelerates the transfer of critical synchronization data from thread to thread, which translates into an average performance benefit of 14% compared to a baseline SMT when running fine-grain, synchronization-intensive workloads. Furthermore, ITSLF also avoids the second search that stores perform (when writing to the L1) in the LQs of other threads in an SMT core. This reduces contention in the LQ snoop port and helps improve the performance of synchronization-poor workloads, where ITSLF outperforms the baseline SMT by 8% on average.

7. REFERENCES

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj 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)*, pages 33–42, April 2009.

[3] Khary J. Alexander, Christian Jacobi Jonathan T. Hsieh, and Martin Recktenwald. Load and store ordering for a strongly ordered simultaneous multithreading core. U.S. Patent US14511408, October 2014.

[4] Jean-Loup Baer. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*. Cambridge University Press, 1st edition, 2009.

[5] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication characterisation of Splash-2 and Parsec. In *Int'l Symp. on Workload Characterization (IISWC)*, pages 86–97, October 2009.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

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

[8] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. Invisifence: Performance-transparent memory ordering in conventional multiprocessors. In *36th Int'l Symp. on Computer Architecture (ISCA)*, pages 233–244, June 2009.

[9] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *2008 Conf. on Programming Language Design and Implementation (PLDI)*, pages 68–78, June 2008.

[10] Alper Buyuktosunoglu, Ali El-Moursy, and David H. Albonesi. An oldest-first selection logic implementation for non-compacting issue queues. In *15th Annual Int'l ASIC/SOC Conference*, pages 31–35, September 2002.

[11] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Conf. on Supercomputing (SC)*, pages 52:1–52:12, November 2011.

[12] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *25th Int'l Symp. on Computer Architecture (ISCA)*, pages 142–153, June 1998.

[13] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven 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)*, pages 105–118, March 2011.

[14] Martin Dixon, Per Hammarlund, Stephan Jourdan, and Ronak Singhal. The next-generation Intel core microarchitecture. *Intel Technology Journal*, 14(3):8–28, March 2010.

[15] Michel Dubois, Murali Annavaram, and Per Stenström. *Parallel Computer Organization and Design*. Cambridge University Press, 2012.

[16] Agner 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`, March 2021.

[17] Andrei 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`, November 2020.

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

[19] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *20th Int'l Conf. on Parallel Processing (ICPP)*, pages 355–364, August 1991.

[20] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Syncron: Efficient synchronization support for near-data-processing architectures. *27th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, February 2021.

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

[22] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *45th Int'l Symp. on Computer Architecture (ISCA)*, pages 46–61, June 2018.

[23] Andrew D. Hilton and Amir Roth. SMT-directory: Efficient load-load ordering for SMT. *IEEE Computer Architecture Letters*, 9(1):25–28, January 2010.

[24] Intel. Intel® 64 and ia-32 architectures optimization reference manual. `www.intel.com`, June 2016.

[25] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. In *44th Int'l Symp. on Computer Architecture (ISCA)*, pages 481–493, June 2017.

[26] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers (TC)*, 28(9):690–691, September 1979.

[27] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman 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)*, pages 694–701, November 2011.

[28] Ching-Kai Liang and Milos Prvulovic. Misar: Minimalistic synchronization accelerator with resource overflow management. In *42nd Int'l Symp. on Computer Architecture (ISCA)*, pages 414–426, June 2015.

[29] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, September 2005.

[30] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[31] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom application transaction processing benchmark. `http://tatpbenchmark.sourceforge.net/`, 2011.

[32] Irma E. Papazian. New 3rd gen Intel® Xeon® Scalable processor (Codename: Ice Lake-SP). In *32nd HotChips Symp.*, pages 1–22, August 2020.

[33] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *41st Int'l Symp. on Computer Architecture (ISCA)*, pages 265–276, June 2014.

[34] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. In *45th Symp. on Principles of Programming Languages (POPL)*, pages 19:1–19:29, January 2018.

[35] Alberto Ros and Stefanos Kaxiras. Speculative enforcement of store atomicity. In *53rd Int'l Symp. on Microarchitecture (MICRO)*, pages 555–567, October 2020.

[36] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, April 2016.

[37] Michael L Scott. Shared-memory synchronization. *Synthesis Lectures on Computer Architecture*, 8(2):1–221, 2013.

[38] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[39] André Seznec. The L-TAGE branch predictor. *The Journal of Instruction-Level Parallelism*, 9:1–13, May 2007.

[40] Tpc benchmark b. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf`, 2010.

[41] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. In *22nd Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 119–133, April 2017.

[42] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 392–403, June 1995.

[43] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.