Temporarily Unauthorized Stores: Write First, Ask for Permission Later

Juan M. Cebrian
Computer Engineering Department
University of Murcia
Murcia, Spain
Email: jcebrian@um.es

Magnus Jahre
Department of Computer Science
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
Email: magnus.jahre@ntnu.no

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

***Abstract*—x86 processors implement a total store order (x86-TSO) consistency model, which requires stores to update memory in a sequenced manner. The latency of stores is then hidden by the store buffer (SB), which holds stores until the write is performed. On a long latency cache miss, however, stores block the SB, eventually stalling the processor and degrading performance. Contemporary industrial high-performance processors deal with this situation by overprovisioning the size of the SB, but this comes at the cost of energy and latency overheads.**

In this work, we remove the stalls caused by stores blocked at the head of the SB while reusing existing processor resources, either improving performance when SB size is kept constant or maintaining performance while reducing SB size. Our proposal, *Temporarily Unauthorized Stores (TUS)*, achieves this by extending the functionality of 1) the write combining buffers, to allow them to coalesce stores while maintaining x86-TSO consistency, and 2) immediately write data to the first-level cache upon a miss (i.e., providing an always-hit illusion) but temporarily keeping the written data invisible to the cache coherence protocol, i.e., these stores are temporarily unauthorized. TUS makes temporarily unauthorized stores visible in x86-TSO order without speculation or rollbacks once write permission is obtained. In essence, TUS logically transforms the write combining buffers and the first-level cache into an "extension" of the SB.

TUS improves performance by up to 26% (3.2% on average) while reducing the total energy-delay-product (EDP) by up to 35.9% (6.4% on average) for SB-bound benchmarks with a 114-entry SB compared to our baseline architecture with an SB of the same size. When configured with a 32-entry SB, TUS yields a performance improvement of 2% over a 114-entry SB baseline while reducing SB energy per search by a factor of 2×, SB area by 21%, and store-to-load forwarding latency from 5 to 3 cycles.

***Index Terms*—Store buffer, write operations, cache coherence protocols, multi-core architectures**

I. Introduction

Modern architectures allow store instructions to be committed immediately from the perspective of the processor core as long as they are eventually made visible to the rest of the system in accordance with its memory consistency model, e.g., Total Store Order (x86-TSO) in all x86 architectures [39]. Decoupling the update of the architectural state from the update of the (shared) memory state enables hiding the execution latency of store instructions. This capability is enabled by the Store Buffer (SB)[1] which keeps stored data after the store instruction commits and until the memory subsystem has been updated. The SB has finite capacity, and, if it becomes full, the processor must stop dispatching stores, which in turn stalls instruction execution. SB stalls have a detrimental effect on performance — to the extent that leading performance analysis approaches strive to capture SB stalls [20], [49].

Overprovisioning is the go-to strategy for enabling high-performance store handling. One example of this trend is that commercial architectures appear to focus on making the SB as large as possible, e.g., Intel has increased the size of the SB from 56 in Sky Lake to 114 in Alder Lake [25]. SBs are typically implemented as content-addressable memory (CAM) structures because they must be associatively searched on every load [16]. Large CAMs are, however, costly in terms of area and energy per access (search), and SB size can hence only be increased so much before its search latency ends up on the critical path and limits the processor's clock frequency. Large SBs also affect *serializing* events that require the SB to be flushed (e.g., a fence). Moreover, since SB search latency increases with the number of entries, there is an unnecessary detrimental performance overhead on workloads that perform well with (much) smaller SBs [3].

Although large SBs incur unattractive overheads, they also yield suboptimal performance because they stall when encountering long-latency store misses (e.g., missing out on a 9.6% speedup for *mcf*) and long store bursts (e.g., leaving a 26.2% speedup on the table for *gcc*). Under x86-TSO, stores must leave the SB (complete) in program order, and a store instruction that misses in the Last-Level Cache (LLC) will hence block the SB for hundreds of clock cycles and eventually incur a stall. With store bursts, the challenge is that the stores enter the backend much faster than they can be drained, which

This work has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (ECHO, grant 819134), from the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (DAMAS, grant PID2022-136315OB-I00), from the MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR (HEEDA, grant TED2021-130233B-C33), Research Council of Norway (grant 286596) and the European Union's Horizon 2020 research and innovation program (grant 101034240).

[1] In this paper, we model a unified store buffer for non-committed and committed store instructions as in x86 processors [24].

also causes the SB to block when the burst is sufficiently long.

Some state-of-the-art proposals [2], [47] scale the SB using additional hardware structures (i.e., perform even more overprovisioning). For example, the Scalable Store Buffer (SSB) [47] augments the SB with a large in-order queue (which buffers both the address and the data of typically 1K stores) and uses the first-level data cache (L1D) to provide store-to-load forwarding. While the 1K in-order queue is much simpler than the CAM-based SB, it still incurs significant area and energy overheads. Moreover, the design complexity of SSB is high because it has to iterate over the in-order queue and re-play the stores upon receiving a data invalidation request from another core. SSB also updates the second-level cache for each store, since it does not perform store coalescing, which further increases its energy overhead.

On the other hand, the Coalescing Store Buffer (CSB) [38] non-speculatively coalesces stores to non-consecutive cache lines while respecting x86-TSO — thereby improving SB utilization and reducing the number of cache accesses compared to SSB. CSB blocks the pipeline on long-latency store misses, since it does not support store-wait-free mechanisms [47]. Furthermore, it requires either a collapsible SB or a separated buffer for pre-commit and post-commit (i.e., coalesced) stores and it is thus challenging to implement. For these reasons, previous works [2], [38], [47] remain hard-to-implement in hardware — and simply increasing SB size is currently the most appealing option for the industry.

Therefore, there is a pressing need for a store handling mechanism that yields both high performance and low overhead, and we introduce *Temporarily Unauthorized Stores (TUS)* to address this need. The key insight that enables TUS is that the latency of store instructions can be hidden as long as 1) the data written by all in-flight stores remain invisible to external requests until write permissions for the required cache lines are obtained, and 2) the updates to memory are (eventually) made visible in program order (to comply with x86-TSO). The challenge is thus to buffer all in-flight store data with minimal area and energy overheads while ensuring that dependent loads are provided with the most recently stored value and memory updates follow program order. TUS addresses these challenges by allowing stores to write in the L1D without first obtaining write permissions. We call these temporarily unauthorized stores which TUS stores in the L1D but keeps invisible to the rest of the system (i.e., they are non-coherent). TUS employs a small *Write Ordering Queue (WOQ)* to track store order (no data) and ultimately make unauthorized stores visible in program order (x86-TSO).

TUS enables local stores to write multiple unauthorized cache lines, allowing stores to coalesce across multiple non-consecutive cache lines *outside of the SB*. We re-purpose the Write Combining Buffers (WCBs), which modern processors use to coalesce stores in code regions for which consistency is not required (non-temporal stores), to coalesce coherent stores. WCBs can be seen then as unauthorized L1D entries. When acting as coalescing buffers, they improve L1D bandwidth utilization and reduce L1D energy consumption.

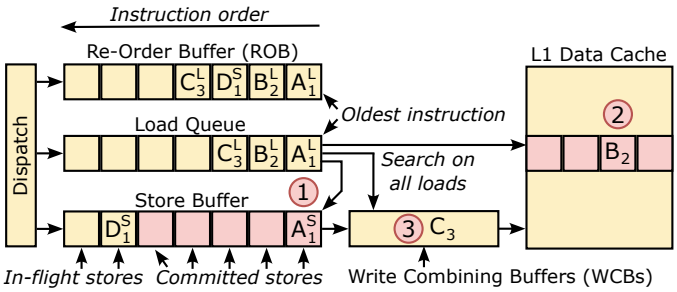

Fig. 1. Executing store instructions in a high-performance processor core

TUS non-speculatively avoids the deadlocks that can arise upon external requests to temporarily unauthorized stores while maintaining low hardware complexity. It enforces 1) a global order on how cores acquire the resources required to make temporarily unauthorized stores visible, and 2) that *non-consecutive* stores that access the same cache line are made visible to the memory system at the same time as those stores between them, i.e., as an *atomic group* [38].

Our evaluation using the gem5 simulator demonstrates that, compared to our baseline processor, and at a storage overhead of only 272 bytes (for the WOQ), TUS improves performance by 3.2% (3.5%) on average for SB-bound SPEC CPU2017 and Tensorflow (Parsec) benchmarks, while reducing the EDP of the processor by 6.4% (5.1%) on average. When configured with a 32-entry SB, TUS yields a performance improvement of 2% on average compared to our 114-entry SB baseline while reducing SB energy per search by a factor of 2×, SB area by 21%, and store-to-load forwarding latency from 5 to 3 cycles.

II. Background

Store buffer primer. The processor front-end fetches and decodes instructions in program order and dispatches them to the functional units and the Re-Order Buffer (ROB). Figure 1 focuses on the key resources for processing store instructions which are the dispatch stage, the ROB, the SB, the load queue, the Write Combining Buffers (WCBs), and the L1D, and shows the state of these resources in a single clock cycle. When a store (load) instruction has been fully decoded, it is dispatched to the SB (load queue) and the ROB in the same cycle. The processor commits instructions when they are fully executed and the oldest instruction in program order. We model a core that can commit up to eight instructions each clock cycle (see Section V for details about the experimental setup).

When a store instruction reaches the head of the ROB, it is committed immediately because committing store instructions only requires setting the "committed" bit in the SB. In Figure 1, the committed stores have a red background and the in-flight stores have a yellow background. The store instruction A_1^S has hence been committed, whereas D_1^S has not (see (1)). Loads leave the load queue upon commit and all instructions leave the ROB when they commit; A_1^S is therefore no longer in the ROB. We label store and load instructions with the cache line they access, and the subscript signifies the index of the

word accessed within the cache line. The superscript marks if the instruction is a load or a store, i.e., X_i^S and X_i^L are store and load instructions, respectively, that access word i in cache line X. We will omit the store and load distinction in examples that only consider stores.

Loads might access recently stored data, and hence the processor must check if the SB has a more recent copy of the data accessed by a load. In Figure 1, the load instruction A_1^L will read the data written by the store A_1^S, and will therefore retrieve stale data if it accesses the L1D. To account for this situation, processors provide store-to-load forwarding by accessing the SB and L1D in parallel on every load. Load A_1^L will hence find the most recent data in the SB since it was recently written by store A_1^S (see ②). This requires hardware support for retrieving the data written by the youngest store to an address, and the requirement to support this operation is the root cause of the poor scalability of SBs. In contrast, load B_2^L does not find any recent writes in the SB and therefore accesses the L1D (see ③).

Write Combining Buffers (WCBs). Processor manufacturers, including Intel [26], AMD [4], IBM [23], and Arm [41], use WCBs to optimize operations that do not have to be consistent (such as memory-mapped I/O), and the number of WCBs has increased from six per core on Intel Core Duo to twelve in Icelake [25]. The use of the WCBs is controlled by marking memory regions as write-combining, either in the kernel, device driver, or application (with permissions). WCBs are located between the SB and L1D, and stores are written to the WCB before they go to L1D. The WCBs hence also need to be searched on every load, and load C_3^L in Figure 1 finds the most recent copy of its data in the WCB ③.

Store prefetching. Prefetching can reduce the number of stores that miss in cache and thereby improve the performance of store-intensive applications without modifying the SB architecture. A plethora of load-focused prefetchers [5], [7], [8], [28], [29], [32], [34], [35], [37] have been proposed, but they need to be conservative when requesting write permission as this can lead to unnecessary invalidations [27]. The most common store-focused approach is to request write permission when a store commits [25], [45]. (Our baseline, described in Section V, includes both a stream prefetcher and requests write permission when stores commit.) Write permission can also be speculatively requested when stores execute [18]. The state-of-the-art store-focused prefetcher is Store Prefetch Burst (SPB) [12] which prefetches write permission of all cache lines in the memory page upon detecting a store burst. Prefetching in general, and SPB in particular, are however limited because (i) prefetchers thrive when access patterns are regular but irregular access patterns are common for stores [6], and (ii) the SB will still block on store bursts with prefetching enabled. (We will demonstrate that SPB yields lower performance than TUS, SSB, and CSB in Section VI).

III. Temporarily Unauthorized Stores (TUS)

This section offers a conceptual description of how TUS provides a coherent, non-speculative, and deadlock/livelock-free way to perform out-of-order cache stores. Further details on how to implement TUS will be discussed in Section IV.

A. Unauthorized Stores

Operation. The key design point of TUS is to provide the illusion that stores leaving the SB always hit in L1D, without additional structures to temporally store the data and without waiting for permissions (unauthorized). The challenge is how to handle *multiple* unauthorized stores, as the mandated store sequence by x86-TSO may be disrupted if permissions arrive out of order. A new structure, the write ordering queue (WOQ), will hold enough information to keep track of the *order* cache lines must be made visible to the rest of the cores to maintain x86-TSO consistency as well as a mask that tracks which bytes have been written for each cache line. This information is required to update the cache line once it is received by L1D.

When the stored data reaches the L1D controller, two things can happen: 1) the cache line does not have write permissions; or 2) the cache line has write permissions. In the first case, the data is written into an L1D entry (evicting an existing one if necessary – although usually an allocated entry from the prefetch-at-commit should be found). A new bit is added for each entry in the L1D to mark cache lines as *not visible* when they write unauthorized data, and a write permission request is sent to the memory hierarchy. While *unauthorized*, the cache line is not visible to the coherence protocol, it cannot be selected for replacement — since there is no other copy of this data — and it cannot be forwarded to other cores — since this would violate x86-TSO. When the memory subsystem has retrieved the cache line with exclusive (write) permission and it reaches L1D, its data is combined with that stored in L1D using the mask, and the WOQ entry is marked as ”ready”, but it is not yet made visible to external cores. Stores will eventually be made visible respecting x86-TSO order as specified by the WOQ. Deadlock and livelock avoidance will be discussed Section III-C. Still, we can advance that it is based on cores deciding who relinquishes write permissions according to a global order that guarantees forward progress. In the second case, the cache line was *authorized* in L1D, meaning it is not in the WOQ. Therefore, an entry in the WOQ is allocated for this cache line. However, if the cache line is modified, an update to the private L2 is sent to keep a valid *updated* copy there. The cache line is marked as ”ready” directly, but not visible (unauthorized) to external cores. Again, stores are made visible respecting x86-TSO order as specified by the WOQ.

In both cases, when the WOQ is full or there is no cache way available for replacement in the said set, the store cannot be serviced. Note that this does not necessarily cause a pipeline stall if the SB is not full. In addition, when the core detects a *serializing* event that requires the SB to be flushed (e.g., a fence), we must also wait for all cache lines to have been made visible to the coherent world in x86-TSO order based on the WOQ information. At worst, it will take TUS the same time to flush both structures since the stores are the same.

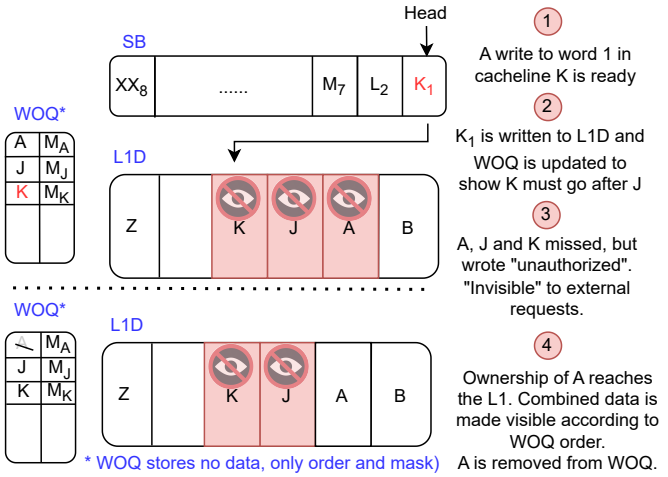

Fig. 2. TUS write path example assuming normal behavior

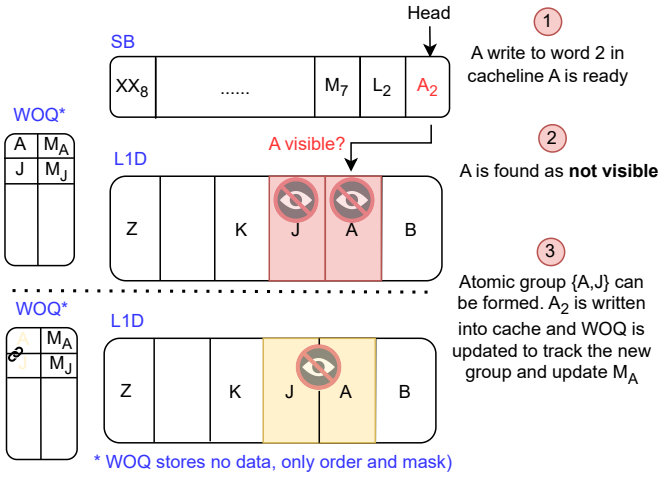

Fig. 3. TUS example with store cycles and 4-way associative L1D

Example. Figure 2 depicts a simplified example of the new store path when using TUS (assuming normal behavior). A write to word 1 in cache line K is committed, and a prefetch requesting write permission is sent to L1D cache (1). K_1 is written into L1D and the WOQ is updated to include K in the store sequence (2). Writes to cache lines A, J and K missed in L1D, but wrote their data as *temporarily* unauthorized. They will remain *unauthorized* until permissions arrive, and be made *visible* (authorized) in the order set by the WOQ (3). Write permissions and data from memory eventually arrive at L1D for cache line A, where they are combined with existing data. Finally, cache line A is made visible to the rest of the cores according to x86-TSO order based on WOQ information (4).

B. Store Cycles and Coalescing

Handling cycles. TUS can handle multiple unauthorized stores by simply tracking their order in the WOQ to respect x86-TSO. However, cycles may appear among the stores, that is, a write targets an old unauthorized cache line that is different than the last one. For example, the sequence of stores ABA forms a cycle, where A must be made visible both before and after B. The only way to do that is to make them visible *at the same time*. In order to allow for cycles we extend the WOQ to support the concept of *atomic groups* [38]. If we can guarantee atomicity concerning external loads and stores, the order of the stores inside the atomic group does not matter, meaning x86-TSO is preserved (see details in Section III-D). Initially, each store will create a new atomic group of a single element and queue in the WOQ, unless a cycle is found. When a store hits in L1D, it checks its *not visible* bit. If it is set, we found a cycle, and an entry must exist in the WOQ. In this scenario, since our goal is to continue writing unauthorized stores, we need to combine several atomic store groups into a single one to respect x86-TSO. Since all previous unauthorized stores have already allocated an entry in L1D, progress in this situation is guaranteed. Further details about deadlock freedom in the presence of limited hardware resources are given in Section III-C.

Example. Let us assume the following sequence of stores that have already been completed (left the SB), A_1, J_1. Figure 3 shows the state of L1D and the WOQ. The WOQ shows that currently, each cache line (A and J) is its own atomic group. When A_2 completes (1), it finds A in "not-visible" state in L1D (2). Since cache line A is already present in the WOQ, adding A at the end will generate a cycle. In essence, A would need to be made visible both before and after J. The only solution is to create a new atomic group $\{A, J\}$. A_2 can safely write into L1D, update the mask at the WOQ (M_A), and mark at the WOQ that both cache lines form an atomic group (3).

Supporting coalescing. TUS also enables coalescing multiple non-consecutive writes[2] to the same cache line while maintaining x86-TSO consistency. To enable this, TUS needs to buffer multiple unauthorized stores while coalescing them, which is similar to the requirements of WCBs for stores that do not require consistency. In order to save resources, TUS will hence not use new hardware to implement these buffers, but extend the WCBs to support temporal stores for non-consecutive writes to multiple cache lines. TUS achieves this by reusing the same hardware used for maintaining multiple atomic groups in cycles.

Starting with empty WCBs, the store at the head of the SB is placed in the first buffer. If the next store at the head of the SB is to the same cache line as the one in the WCBs, then it coalesced in the same buffer. If not, it will try in the next buffer (this allows for shuffling and interleaving of stores across n unique cache lines). If it writes to an existing WCB different than the last one it wrote, there is a cycle and the WCBs must be treated as an atomic group. The resulting combined store group from multiple buffers cannot exceed the associativity of the cache in any given set used in the group. This means that when stores write to L1D, TUS can only proceed if there are available ways for *all* of them in L1D. This is the same

[2]Physical addresses must be valid to perform coalescing, e.g., TUS does not coalesce stores due to interrupts and programmed input/output.

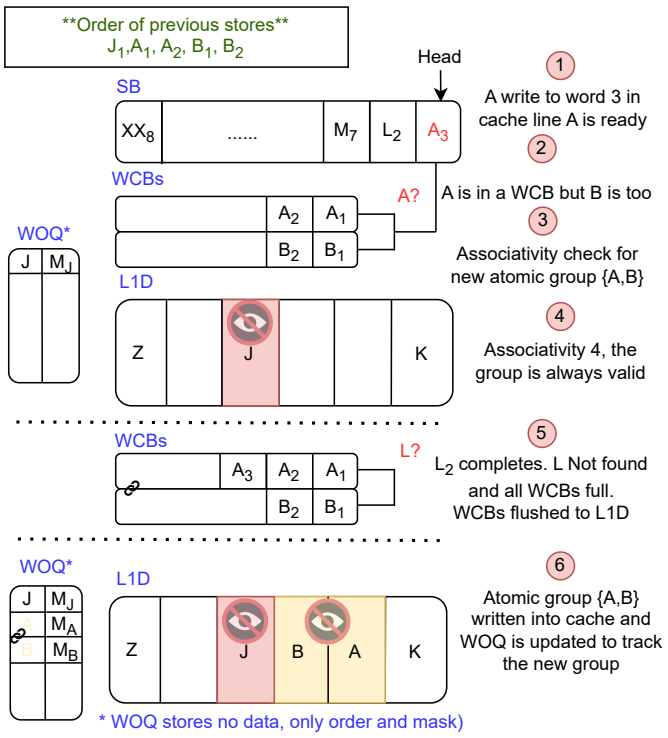

Fig. 4. TUS coalescing example with 2 WCBs and a 4-way associative L1D

restriction we had previously but extended to n cache lines per atomic group. Therefore, all checks involving the atomic group must be performed to all its cache lines as a whole, otherwise, we cannot guarantee deadlock freedom when they reach the WOQ.

Example. Figure 4 shows an example of how using atomic groups can coalesce writes independently. Let us assume the sequence of committed stores A_1, A_2, B_1, B_2, A_3, L_2. Upon completion, A_3 searches the WCBs to find cache line A (1). A is found in a WCB, but there is another WCB in use, so a cycle is possible (2). Since the last written WCB contained B and the new one is A, there is a cycle. The atomic group $\{A, B\}$ is formed in the WCBs. Now L_2 completes and searches for L in the WCBs. L is not found and there are no available WCBs, so they must be written into L1D as unauthorized (5). The atomic group $\{A, B\}$ can use, at worst, two ways of the same set. If we assume that the associativity of the system is 4, this atomic group can always fit in any given set (3) (4). A and B are written into L1D, the WOQ is updated with the new atomic group (6). Note here that J remains its own atomic group since the cycle only affects A and B, and according to the order in the WOQ, J *will always be made visible first*.

C. Dealing with External Requests

Avoiding deadlocks. To maintain x86-TSO consistency, local stores must be made visible to other cores in the same order as in the program. Our proposal achieves this goal by making cache lines *visible* in the (atomic) order they are placed in the WOQ. However, cache lines in TUS can be ready (we have permissions and data has been combined), but *not-visible*, since there may be an older atomic group in the WOQ that is not ready yet. This could easily lead to a deadlock scenario when two cores have atomic groups that are not disjoint and both have obtained permissions over *some* of the cache lines. To avoid deadlocks, TUS decides which core will relinquish cache line(s) in a way that guarantees forward progress.

To do so, we will rely on the lexicographical (lex) order of memory addresses [14], [38]. The lex order sets a global *sub-address* order to each cache line and is dictated by a set of bits that are used to map it on a resource. We represent that order using alphabet letters in our examples. By acquiring write permissions for *all* cache lines older than the conflicting store in lex order, including older atomic groups, we guarantee forward progress. This means that multiple cache lines that we have acquired permissions for, and have already combined, can lose permissions. When an external request is received, TUS disables the creation of new cycles that involve that atomic group, so that the lex order does not change due to newly completed stores trying to leave the SB. While cycle support is disabled, the store at the head of the SB that creates the cycle is not allowed to complete. The core that violates lex order will relinquish permissions for all cache lines with lesser lex order than the conflicting store, but only for *older* stores (based on WOQ order). A new write permission request will be created when the store has the lowest lex order of all older stores in the WOQ. This adds another restriction when combining groups, i.e., a *lex conflict* (two cache lines that share the same lex order) in a group is not allowed. In that case, coalescing is disabled and the store is not attended until the lex-conflicting store has been made visible.

Lex conflicts and limited resources. Lex conflicts serve as an ordering point for stores, by preventing the creation of atomic groups that could compromise forward progress due to limited resources in the cache hierarchy. Disallowing lex conflicts within atomic groups simplifies the reasoning and mechanisms required to achieve forward progress. Otherwise, when conflicts are allowed [19], new messages need to be introduced in the memory hierarchy.

The choice of a proper sub-address has been studied in previous works [19], [38]. We opt for choosing the 16 less significant bits of the cache line address, which are also the same number of bits used to index the directory (and the LLC). This way, once exclusivity of the cache line is obtained to write an atomic group, it is guaranteed that no other cache line within the same group would require an entry in the same directory set. If several cores performing atomic writes have lex-conflicting writes, those acquiring an entry in the specified set are guaranteed to complete. When an atomic group is made visible, new space will be available to be replaced in the shared resources for other conflicting writes (as directory-induced invalidation will not be delayed once the atomic group write completes). Hence, no deadlocks because of a directory-induced invalidation can occur when (i) using the aforementioned lex order and (ii) preventing the presence of conflicts in a group.

When a store needs to perform an authorized write in a

modified L1D entry, the current written data needs to be updated to L2. Since our L1D and L2 are inclusive, the L2 always has a copy of the cache line, thus guaranteeing progress. In addition, the L2 cannot choose the eviction of a cache line that preserves lex order, the invalidation request will be NACKed if it preserves lex order, forcing the L2 to "refresh" the replacement policy and ask for another cache line to be replaced (similar to [19]). The associativity restriction when forming atomic groups guarantees there will be several entries that can be replaced.

Example. Figure 5 shows an example of how TUS resolves deadlocks. Let us assume core 0 has obtained the write permission and modified cache line C in L1D. This cache line is no longer unauthorized, it is modified (dirty), but not visible yet (1). In order to become visible, it requires both that cache line R is made visible (WOQ order), and cache line D to be ready, since it is part of the same atomic group. Meanwhile, core 1 has obtained write permission and modified cache line D in L1D (1), and is in a similar situation with cache lines P and C. At some point, core 0 L1D will receive an invalidation coming from its L2 for cache line C forced by a write permission request from core 1 (2). At the same time, the L1D of core 1 receives an invalidation coming from its L2 for cache line D forced by a write permission request from core 0 (2). Neither of the cores can remove the cache line from their L1D since there is no other copy of the valid, authorized, but not yet visible data in the core. However, if they do not reply while waiting for the stores to become visible, we have a deadlock scenario (3).

The solution is to make one core proceed and ensure that the other core relinquishes its write permissions and waits. We use the lex order to ensure that all cores in the system agree upon which core(s) shall relinquish write permissions and let other(s) proceed. First, we disable the formation of new cycles that involve the atomic group $\{C, D\}$, to prevent changes in the existing atomic groups. We then check which core has acquired write permissions for all cache lines with lesser or equal lex order than the conflicting cache line. Core 0 checks the lex order to see if it owns all cache lines with lex order lesser than C (4). Since it does, it *will not* give away the permissions, and make core 1 wait (5). In contrast, the lex order check of core 1 reveals that another core wants permissions to D, while it is still waiting for permissions for C, which has a lower lex order (6). In this case, core 1 gives away the permissions for D, which then becomes unauthorized again, but keeps the modified data for D in its L1D cache (7). Core 0 will be provided with the old, unmodified, version of D that is stored in the private L2 (8). This will allow core 0 to proceed, thereby removing the deadlock.

Core 1 however needs to get write permissions for D at some point, and, once permissions are acquired, combine the updated data from core 0 with its unauthorized write. To prevent continuous exchange of write permission requests and invalidation requests, core 1 will re-send the write permission request for D only when the cache line is the lesser-most address in lex order in the atomic group at the head of the

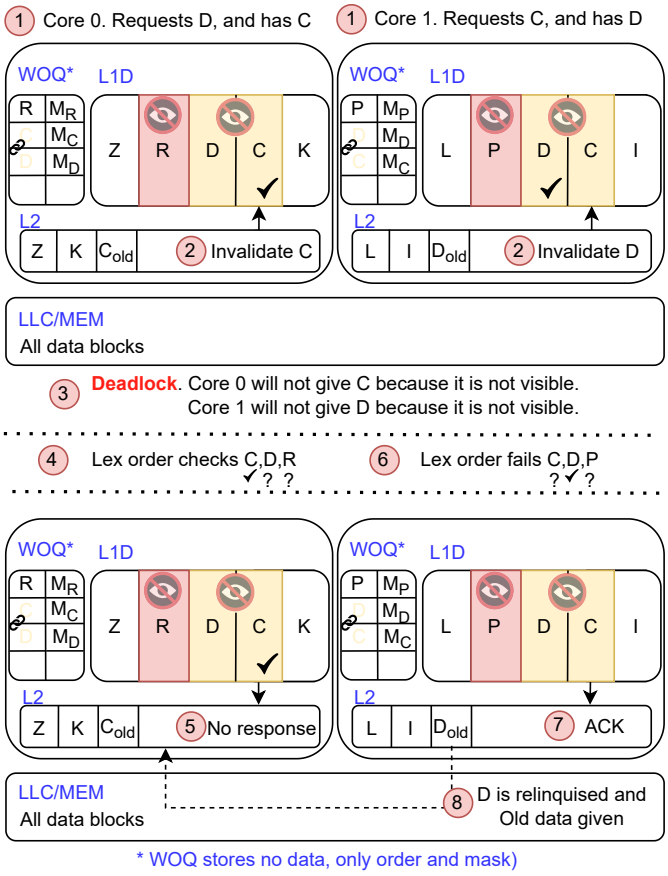

Fig. 5. TUS example of dealing with external requests with lex order

WOQ. In our example, the order is shown at (6), and the new request will be issued when (i) core 1 obtains permissions for C, and (ii) P has been made visible.

D. TUS preserves TSO

TUS preserves all TSO orders. Conceptually, unauthorized stores can be considered as part of the SB, and making them visible is akin to leaving the SB in a standard TSO implementation. As stated by Ros and Kaxiras [38], allowing coalescing can violate TSO with respect to loads and stores. However, when (i) coalesced stores are written atomically and (ii) loads are delayed until those stores complete their write, TSO is preserved. Similarly, next, we elaborate on how each TSO order is preserved in TUS:

- Store→Store: Stores are made visible to other cores in program order, or atomically when they coalesce. Hence, no other core can see two stores performing (made visible) out of order.
- Load→Load: Loads can execute out of order, but the reordering is only visible by other cores if they write to the same load addresses. Typically, TSO squashes loads when the accessed cache line loses read permission, by snooping the load queue [18]. For an unauthorized store to be made visible, it requires exclusive permission, squashing remote loads to the same cache line in the process. While writing atomically, remote loads to the

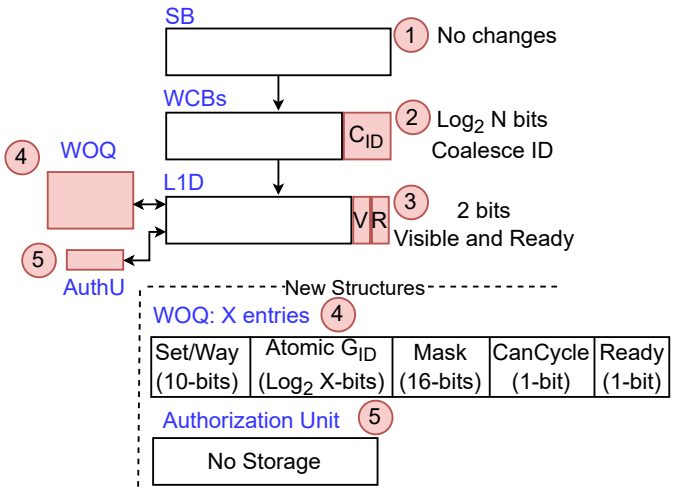

Fig. 6. TUS hardware changes

involved cache lines are delayed. Hence, load→load reordering cannot be observed.

- Load→Store: Loads perform before committing and stores perform after committing. Since commit happens in program order, the load→store order is always enforced.

IV. Implementation of TUS

Hardware support. Figure 6 describes TUS' main hardware units. TUS does not require any changes to the SB (1), but it requires a set of buffers to coalesce stores that include a mask to identify the bytes that have been written in the cache line; TUS exploits WCBs for this purpose (2). In addition to the data and the mask, TUS requires $\log_2 N$ additional bits per entry, where N is the number of buffers used for coalescing, to identify which buffers belong to a coalesced (atomic) group (C_{ID}). We empirically determined that two buffers are enough for our applications and system configuration, so only one extra bit per WCB is required. L1D entries need to be extended with two bits (3), The first bit is used to mark the entry as not visible. The second bit marks it as ready, that is, it has acquired write permissions and combined the new data with the one that came from memory.

Overhead. The WOQ can be implemented as a circular buffer that stores the necessary information to track the order in which unauthorized cache lines must be made visible to the coherent world (4). In contrast to the SB, the WOQ is not on the critical path. It is also smaller than the SB and searched less frequently, i.e., the WOQ is searched on store accesses that hit in the L1D and when data or invalidation requests are received from the L2 whereas the SB must be searched for every load instruction. Moreover, we search the WOQ to find 10-bit tags that yield substantially lower logic complexity than the 64-bit virtual addresses.

Each entry in the WOQ records: 1) The location of the cache line in L1D in the form of set/way. For a 48KB L1D with an associativity of 12, 10 bits are required. 2) The atomic group id ($AtomicG_{ID}$). Since each cache line written starts as its own atomic group, we require $\log_2 X$ for an X-entry WOQ. We have determined that 64 entries are a cost-effective size for the WOQ, so 6 bits are required. 3) A mask to track the written bytes in the cache line. For the sake of simplicity we currently only track 32 and 64-bit stores when coalescing. Therefore, 16 bits are required for a 64B cache line. 4) One bit that marks a WOQ entry as unable to participate in a cycle. This is used to prevent changes in the atomic groups when resolving a conflict using the authorization unit. 5) A bit that marks an atomic group as ready to be visible. This bit is set when the data from L1D is combined with the one coming from the memory subsystem. It is cleared only if we lose permissions due to relinquishing a cache line to an external request. The total storage overhead of each WOQ entry is 34 bits, and the storage overhead of our default WOQ is 272 bytes (i.e., 34 $\times$ 64 = 2,176 bits). Finally, we require a simple circuit to determine (lex) order, the authorization unit (5), which yields no storage overhead.

TUS operation flow. Next, we describe a flow chart that shows the interaction between the different hardware structures (Figure 7). We focus on a single write, but N writes with the same C_{ID} will behave similarly, just requiring to perform all the steps as an atomic group, that is, if one write fails a test, all fail. Stores coalesce in the WCBs until they are ready to write into L1D. If the cache line is not found in L1D, the cache controller will try to allocate a way in the corresponding set. If all ways are blocked, either due to cache locks or invisible cache lines or the WOQ is full, the write is not allowed (1). If not, a way is reserved, and the data is written and marked as not ready and not visible. In addition, an entry in the WOQ is allocated at its tail, and the pair set/way is pointed to the correct location in L1D. The mask is stored in that WOQ entry, and a new $AtomicG_{ID}$ is assigned (note that this ID is different than the C_{ID} from the WCBs). The bit *CanCycle* is set to true and the bit *Ready* to false (2).

On the other hand, if the cache line is found in L1D, it can be visible or not. If the cache is found as visible it means it is no longer at the WOQ. If there is no room in the WOQ, the write is delayed. Since this is now the most up-to-date coherent copy, we first update the copy in L2 (3), and then proceed to write the new data, this time marking the cache line as ready, but not visible. A new entry in the WOQ is allocated as stated previously (2). If the cache is found as not visible, it means we found a cycle, and we need to search the WOQ using its set and way (10 bits). When the entry is found, its $AtomicG_{ID}$ must be copied to all entries between itself and the tail of the WOQ, and its *ready* bit is set to false. Please note that there is no possible deadlock due to resources since those cache lines are already allocated in L1D and contain actual data.

When the write permissions and data arrive (4), the WOQ is searched with the set and way of the L1D entry to retrieve the mask. The entry is set to ready in both the L1D and the WOQ, and the data is combined using the mask. If the entry belongs to the oldest $AtomicG_{ID}$ (the one at the head), we check if all entries from the $AtomicG_{ID}$ in the WOQ have their *ready* bit set. If they have, we access each L1D entry

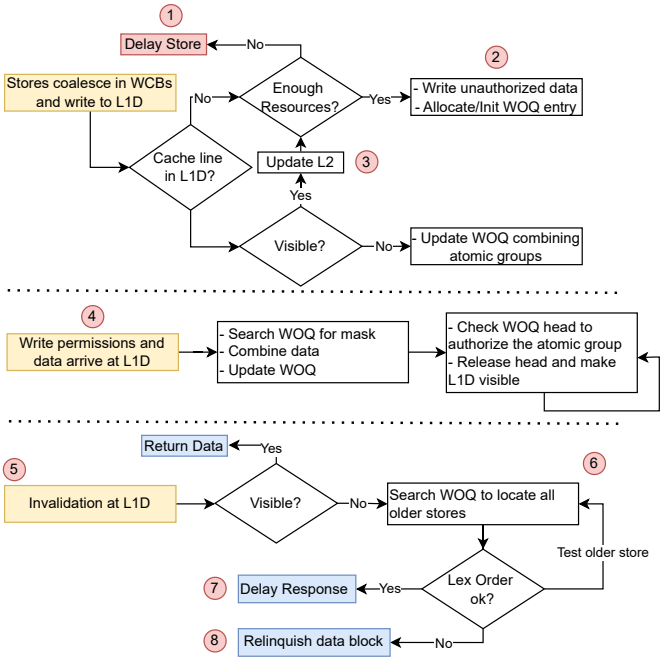

Fig. 7. TUS operation flow

pointed in the WOQ and set them as visible, moving the head of the WOQ to the next $AtomicG_{ID}$. We repeat this process iteratively until we find an $AtomicG_{ID}$ that is not ready.

If an invalidation arrives at the L1D, it checks its visible bit (5). If it is visible it returns the data as usual, since that cache line is both coherent and consistent. If it is invisible, we search the WOQ for the entry and set its *CanCycle* bit to false. Then, all cache lines pointed by the WOQ from that entry to its head need to be checked in the authorization unit for their lex order (6). If the core has permissions for all addresses with lex order lesser or equal than the requested cache line it delays the request, since there is no risk of deadlocks (7). On the contrary, for each entry in the WOQ that does not respect the lex order, it will mark the WOQ entry as not ready, the corresponding cache line as not ready, and change its coherence state to "retry", relinquishing the cache line (8). It will also reply to the L2 with a new packet that acknowledges the invalidation but instructs the L2 to return the unmodified version of the data that it contains. When an $AtomicG_{ID}$ that contains a bit of *CanCycle* set to false reaches the head of the WOQ, we re-check the authorization unit to see if the ordering is now respected. If it does, permissions are requested again, and this time the permission request is guaranteed to succeed.

Other considerations. Store-to-load forwarding can be realized at L1D, but only for those data elements specified by the mask stored in the WOQ. This however requires an associative search of the WOQ to find the entry that contains the mask, or alternatively, add a mask field to each L1D entry. We did not observe any meaningful performance improvement from enabling store-to-load forwarding at L1D, because the store was previously in the SB and performed store-to-load for some time; we hence disabled store-to-load forwarding at the L1D.

TABLE I
CONFIGURATION PARAMETERS

Branch predictor	64KB L-TAGE + ITTAGE [40]
TLB	64-entry L1 and 2048-entry L2
Front-end width	8 (fetch), 6 (decode), 6 (rename) instr.
Back-end width	12 (dispatch), 12 (issue), 8 (commit) instr.
Physical registers	332 integer + 332 floating point
Load/store queue	192/114 entries
Re-order buffer	512 entries
Functional units	1 Int ALU + 3 Int/FP/SIMD ALU
Instr. latency (int)	add (1c.), mul (4c.), div (12c.)
Instr. latency (fp)	add (5c.), mul (5c.), div (12c.)
L1I	32KB, 8-way, 1-cycle latency, 64 MHSRs
L1D*	48KB, 12-way, 5-cycle latency, 64 MSHRs, stream prefetcher (stride)
L2*	1MB, 16-way, 16-cycle round trip, 64 MSHRs
L3*	64MB, 16-way, 34-cycle round trip, 64 MSHRs
DRAM	160-cycle latency
(*)	Write-Allocate/Write-Back. L1D and L2 inclusive

Loads will therefore be aliased to the cache line address and will be serviced later when the write permission arrives.

Some programming languages overwrite application code at runtime. In this scenario, our core generates requests to cache lines to L1D for writing and to L1I for reading. We decided to prioritize L1I over L1D, so the pipeline will stall on stores that have been requested by L1I. We can simply set the *CanCycle* bit to false in order to force this cache line to be visible as soon as possible. These entries cannot participate in cycles, since they must be evicted from L1D to be moved to L1I.

V. METHODOLOGY

Simulator. We use *gem5* [10] to model an x86 full-system environment in which we simulate both single- and 16-core processors using gem5's detailed out-of-order core model and detailed memory hierarchy. The simulated system runs Ubuntu 16.04 with Linux kernel version 4.9.4. Our design assumes a three-level cache hierarchy with private L1I/L1D and L2 caches and MESI coherence protocol. Our baseline configuration includes three important modifications to mainline gem5: (1) a prefetch at commit mechanism for stores [24], [45], (2) support for pipelined L1D accesses for stores and (3) varying store-to-load latency based on SB size (5 cycles for 114, 4 for 64, and 3 for smaller sizes as reported by Fog [15]). These modifications significantly improve the rate at which store operations are performed. We also rely on Fog to model execution and issue latencies for instructions that match real hardware. Table I summarizes our main simulation parameters chosen to resemble a modern processor.

Energy consumption is evaluated with McPAT [31] assuming a 22nm technology node (minimum available in the current version), a voltage of 0.6V, and the default clock gating scheme. We incorporate the changes suggested by Xi et al. [48] to improve the accuracy of the models, and faithfully model the overheads incurred by the extra resources and tables required by TUS. The energy overhead of TUS is dominated by L2 updates when a second write is performed over a visible value that is only available in L1D. The WOQ area requirements are 13× smaller than the 114-entry SB

while using 10× less energy per search when the WOQ is implemented as a CAM. Also, the WOQ is searched on store accesses that hit in the L1D and when data or invalidation requests are received from the L2 whereas the SB must be searched for every load instruction. The WOQ is 13× smaller and uses 5× less energy per search than a 32-entry SB. Configuring a 32-entry SB reduces energy per search by 2× and area by 21% compared to an SB with 114 entries.

Benchmarks. For sequential applications, we run all benchmarks with reference input sets from SPEC CPU 2017 compiled using GCC 5.5 with -O2 optimization flags as well as TensorFlow benchmarks from the BigDataBench suite [17]. For benchmarks with multiple input sets, we identify the selected input set by appending an integer identifier to their name (e.g., *502.gcc1* is *502.gcc* with input set #1). We define SB-bound applications as those that show more than 1% of SB-induced stalls for our baseline configuration. We simulate 10 simulation points for each application [42]. Each simulation point runs for 2 billion instructions after a brief warm-up period of 200 million instructions. We include performance figures that cover the geometric mean for *All* benchmarks. However, for the sake of clarity, our detailed evaluation shows per-application results for SB-bound benchmarks.

For parallel workloads, we run a subset of PARSEC-3.0 [9] multi-threaded benchmarks compatible with our simulator using 16 threads and *simsmall* inputs. We measure performance within the region of interest (ROI), including all instructions executed after initialization and before output generation. Statistics are gathered after 100 million cycles within the ROI to warm up the caches. We run 10 seeds per application and average the 3 fastest executions for our evaluation.

State-of-the-art approaches. We compare TUS to the following previously proposed mechanisms, which also aim at improving performance by alleviating SB bottlenecks:

- The **Scalable Store Buffer (SSB)** [47] extends the SB size by using a 1K-entry in-order queue (called TSOB). We implemented an idealized version of SSB that considers a "magic" 0-cycle recovery for invalidations and hence provides an upper bound on the performance that SSB can achieve.
- The **Coalescing Store Buffer (CSB)** [38] allows coalescing non-consecutive store operations before initiating the write to L1D. Our CSB implementation uses the WCBs to perform coalescing. However, when a WCB that needs to write to the cache suffers a miss, the SB stops draining.
- The **Store Prefetch Burst (SPB)** [12] aggressively prefetches a full 4KB page when it detects that stores are filling consecutive cache lines. Still, on cache misses, the SB stops draining, which may happen frequently in applications with an irregular write pattern.

VI. Evaluation

In this section, we compare our proposal (TUS) to other state-of-the-art proposals, SSB [47], CSB [38], and SPB [12] in terms of both performance and energy-delay product (EDP) for sequential and parallel workloads. We also analyze how the different mechanisms affect SB-induced stalls. A design space exploration (DSE) was performed to determine the configuration parameters for TUS which led us to configure two WCBs and a 64-entry WOQ, as well as set the maximum length permissible for a cycle to 16 (that is, the maximum number of cache lines in an atomic group). For sequential applications, after 8 there is no significant difference or drawback. However, for parallel applications, we found out that allowing big atomic groups is detrimental for some applications due to serialization, while others benefit from additional locality.

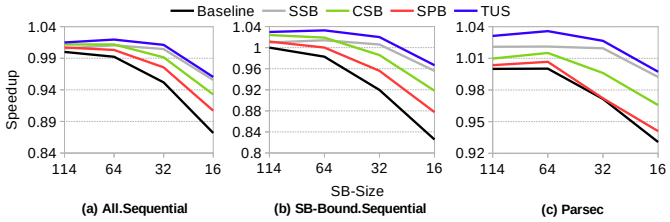

Fig. 8. Scalability analysis with different SB sizes and benchmark suites

Figure 8 shows a scalability analysis with SB size for the different workloads analyzed in this study. Please note that while SSB is included in the figures, it incurs (significantly) higher area and energy overheads than the other proposals, primarily due to its 1K-entry in-order queue (i.e., the TSOB) and direct writes to L2. TUS yields higher performance than SSB, CSB, and SPB regardless of SB size (up to 3.5%), and more importantly, with an SB of 32 entries, it still outperforms the baseline configuration with 114 entries. In addition, we see performance improvements relative to all state-of-the-art proposals when reducing SB size from 114 to 64 entries. Having a smaller SB allows for a faster CAM search time and a one-cycle reduction of the store-to-load forwarding latency (i.e., from 5 to 4 cycles). The key takeaway from Figure 8 is thus that by handling stores in a way that minimizes the performance impact of long latency stores and store bursts, TUS achieves better performance than the SB-overprovisioning strategy used in commercial cores, SSB, CSB, and SPB across all SB sizes.

A. Detailed Analysis of Single-Threaded Workloads

Figure 9 shows the SB-induced stalls, sorting benchmarks based on this metric. Please note that dispatch stalls can be due to different resources, such as ROB, reservation stations, or load queue. The stall is only attributed to the first resource that is missing, and they are not disjoint. Therefore, some SB-stalls may not be accounted for if they overlap with another resource and vice versa. Nevertheless, we see a clear trend for TUS reducing the overall stalls from 6% to 2% on average.

Figure 10-left shows an S-curve of the performance potential for all the analyzed benchmarks (SPEC, Parsec, and TensorFlow) with a 114-entry SB. TUS demonstrates performance gains exceeding 1% in 21 applications, achieving improvements of up to 26%. Figure 10-right shows execution time speed-up over our baseline core design and a 114-entry SB for SB-bound benchmarks sorting benchmarks by SB-

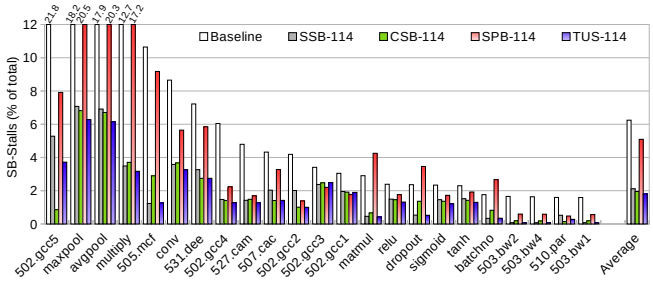

Fig. 9. SB-induces stalls (% of total Cycles) for a 114-entry SB running single-threaded SB-bound applications. Lower is better

induced stalls. We can see a clear dominance of TUS over the state-of-the-art proposals, with no negative side effects on SB-bound applications and an overall gain of 3% over the baseline. SSB achieves 0.9% gains, followed by CSB with 2.4% and 1.1%. The highest gain for TUS comes from *502.gcc5*, with an improvement of 26.1%, and the lowest from *503.bw2* with no performance gain. *502.gcc5* benefits mostly from coalescing, given that CSB and SPB perform similarly to TUS and both still write in x86-TSO order. Checking the store instructions that cause the stalls we see they belong to a store burst. It is also worth highlighting the behavior of *505.mcf*. Coalescing (CSB) and prefetching (SPB) do not help much to reduce SB-induced stalls, but the over-provisioning provided by SSB does. This means that *505.mcf* is dominated by long latency stores, that are captured and hidden by TUS. SPB is having trouble matching the store access patterns on TensorFlow kernels. The execution stalls for SPB while L1D misses are pending increases by 32%, while stalls while L2 misses are pending increase by 41%. This means SPB is being too aggressive and affecting loads. Overall TUS combines the performance gains of CSB and SSB with a simple and elegant design, without the speculative cache pollution caused by SPB.

Figure 11 reports normalized EDP for the same benchmarks and configurations as shown in Figure 10. SSB *increases* the total EDP of the system by 5.9%, while CSB reduces EDP by 6.1%. TUS improves over these proposals, reducing EDP by 6.4%. The hardware resources of SSB are much higher than those of TUS, and they are required to write to the coherent cache (LLC) on every store, while TUS only requires sending updates to the L2 whenever a write hits in L1D and becomes unauthorized. This energy overhead has been accounted for in our results. In addition, TUS reduces the total number of writes to L1D by a factor of 2×, with a peak of 5.5× for *502.gcc5*. This reduction is almost identical in CSB, and both proposals manage to reduce the dynamic component of the energy used by the memory subsystem. Over-provisioned designs, such as SSB or increased SB sizes, harm EDP. Moreover, TUS is the only approach capable of addressing both long-latency stores and store bursts. On applications with SB-induced stalls due to long latency stores (*505.mcf*), TUS improves EDP by 10%.

B. Detailed Analysis of Parallel Workloads

Figure 12-left shows the performance speed-up relative to the 114-entry SB baseline for the multi-threaded Parsec benchmarks. Normalized performance shows up to 17.1% improvements (3.2% average) for TUS, outperforming SSB (2.2%) and CSB (1%). *Dedup* has both bandwidth issues, which are addressed by CSB and TUS, and long latency stores, that can be hidden by both SSB and TUS; SPB addresses neither issue. *Ferret* is dominated by bursts of interleaved stores. Finally, we have *streamcluster*, where SPB is the best-performing state-of-the-art approach. This means that SB-related stalls are related to bursts, and the difference in performance comes from an increase in cache temporal locality. Indeed, TUS retains stores in L1D until permissions are obtained, while SPB will continuously prefetch more cache lines, and both loads and other prefetches can replace the already prefetched lines. TUS reduces the number of stalls while L1D misses pending for *streamcluster* by 27% over SPB.

Finally, Figure 12-right shows the EDP results for Parsec. TUS manages to improve on CSB with EDP improvements of 5.1% (over 2.4%). This is clearly shown in applications such as *dedup* and *ferret*, where coalescing significantly reduces the dynamic energy component of the memory subsystem with a reduction in write accesses of 4.9× and 2× (average of all Parsec is 2.1×). The speedup achieved in *streamcluster* is the main reason for its overall reduction in EDP.

C. Reducing SB Size while Maintaining Performance

In this section, we analyze how TUS behaves under high SB utilization. To do so, the SB size is reduced to 32 entries. A smaller SB reduces the time and energy required to perform CAM searches for store-to-load forwarding. Furthermore, on processors that support simultaneous multi-threading (SMT), the effective size of the SB is divided by the number of hardware threads as the SB is statically partitioned across threads (Section 2.6.9 of Intel's optimization manual [24]). This partitioning is related to the consistency model, and in particular, to *store atomicity semantics* as dictated by the read-own-write-early-multiple-copy atomic model (rMCA) that is followed in actual x86-TSO implementations [1], [46].

Figure 13-left shows the performance S-curve for all evaluated applications, normalized to a baseline with a 32-entry SB. TUS achieves a peak gain of 36.6% in performance, with 21 applications improving over 5%. Figures 13-right and 14-left show the detailed speedups normalized to a 32-entry SB. TUS maintains its performance gains even with a reduced SB size, with average gains of 10.1% for single-threaded SB-bound and 5.8% for Parsec. Figures 15 and 14-right show the EDP improvements for single-threaded SB-bound and Parsec, respectively. TUS improves EDP by 15.7%, followed by CSB with 12% and SSB (5.2%) on single-threaded SB-bound. For Parsec, TUS gains 10.2% followed by SSB with 7.4%.

TUS can achieve better performance than the 114-entry SB baseline using only 32 entries and two WCBs. It does so without causing cache pollution, unlike speculative proposals such as SPB. Note that modern processors implement a store prefetch-at-commit [24], [45] strategy, (+15% performance over the default gem5). Since cache lines have already been

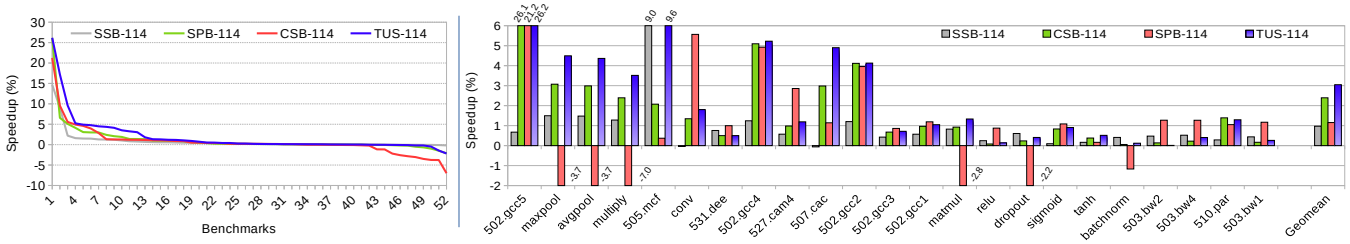

Fig. 10. Speedup S-curve for all applications (left) and single-threaded SB-bound breakdown (right) normalized to a 114-entry SB. Higher is better

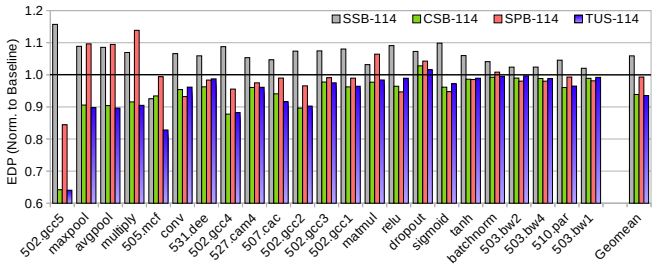

Fig. 11. Normalized EDP to 114-entry SB for single-threaded SB-bound applications. Lower is better

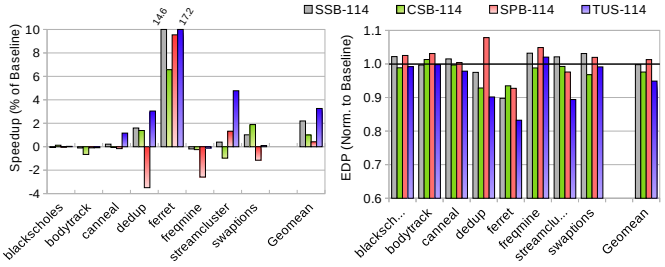

Fig. 12. Speedup (left – higher is better) and EDP (right – lower is better) normalized to a 114-entry SB (Parsec)

replaced by prefetch requests in L1D when TUS writes, it just re-uses those allocated cache lines to store unauthorized data. L1D hit rate goes from 96.36% on the baseline to 96.33% under TUS for a 32-entry SB. cactuBSSN and mcf are the most load-bound applications in SPEC 2017 (86.3% and 88.1% load hit rates with an SB of 32 entries), both included in our results. The next one is xalancbmk, with 91.5%. The hit rates with TUS are 86.39%, 88.2%, and 91.4% for the same applications. More importantly, the number of cycles the pipeline is stalled while there are L1D misses pending drops from 7.2% to 5.8%. This statistic defines the level of memory-boundness of an application [33].

VII. Related work

Masked writes. TUS can reuse the WCBs to perform write coalescing. WCBs include dirty bits to know the part of the cache line that has been modified to allow non-temporal stores to combine their data with the one coming from memory [25]. TUS extends this functionality to allow temporal stores to write partly cache blocks to L1D. In addition, most modern vector support, such as AVX512 and SVE, already support masked writes to L1D [25], [43].

Atomically writing a group of stores. TUS requires making a set of written cache lines atomically visible to memory. A large body of work has proposed solutions towards this goal [13], [19], [21], [22], [30], [36], [38], [44], [47].

The Oklahoma update [44], "all-or-nothing", is a pioneering approach to performing a set of writes atomically. The proposed solution is speculative, which in case of conflicts, rolls back execution. Being the precursor of transactional memory (TM) [22], [30] it also suffers from livelocks. Commonly, livelocks in TM solutions are addressed by using aging mechanisms and a global lock as a backup solution.

Hardware transactional memory (HTM) implementations [21] include support for making a set of writes atomically visible. Essentially, transactional stores speculatively write to the L1D. Transactions can only be committed if the core has write permission for all cache lines involving writes. Stores are then made visible by resetting in bulk a transactional bit stored along with the cache lines. The L2 holds the write permissions and the unmodified copy of data as a backup in case the transaction aborts. We borrow that mechanism from HTM implementations, but without the need to abort the atomic write when losing write permission, since we do not require write permission to allocate the written data in cache. In case of requests from other cores, the L2 data is used.

Similarly, using speculation and rollback-on-conflict mechanisms, BulkSC [13] and ASO [47] aim for enforcing Sequential Consistency while allowing load/store re-orderings within runtime-defined atomics regions. The main difference with TUS is that our approach does not need to roll back on conflicts, i.e., TUS performs writes non-speculatively, saving the storage and complexity required by the speculative state.

CSB [38] can coalesce non-consecutive stores while guaranteeing x86-TSO. Non-consecutive store coalescing can violate x86-TSO if the stores between the coalesced ones do not update memory atomically. By using a global (lexicographical) order and restricting coalescing, CSB can perform the atomic memory updates in a non-speculative way. TUS borrows the global order concept to also perform non-speculative updates. Unlike TUS, CSB requires write permission to update memory, and TUS therefore outperforms CSB (see Section VI).

Using a predetermined global order, other works have proposed hardware multi-address read-modify-write atomic operations [19], [36]. However, these approaches require adding new architectural instructions to read and write multiple cache lines atomically. Again, as with CSB, write permission is also required to execute these multi-address atomic operations.

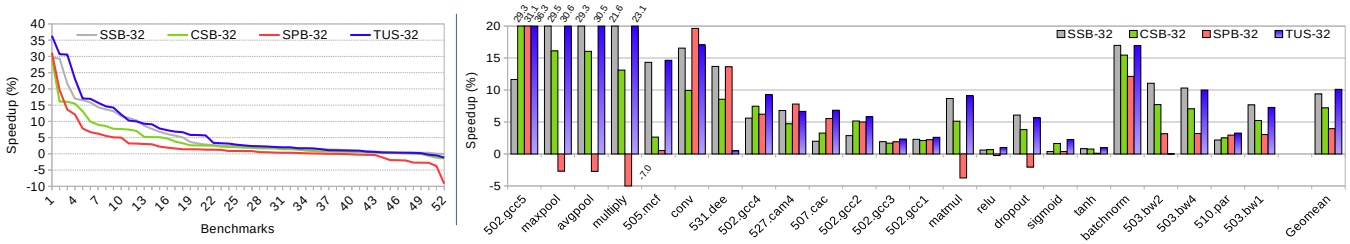

Fig. 13. Speedup S-curve for all applications (left) and single-threaded SB-bound breakdown (right) normalized to a 32-entry SB. Higher is better

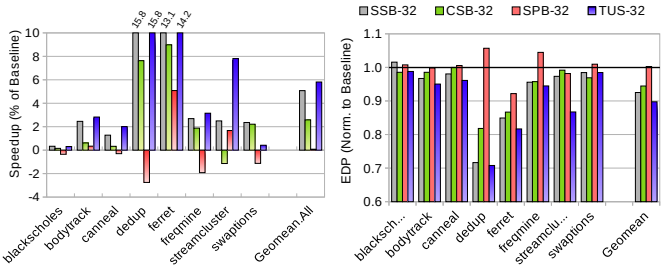

Fig. 14. Speedup (left – higher is better) and EDP (right – lower is better) normalized to a 32-entry SB (Parsec)

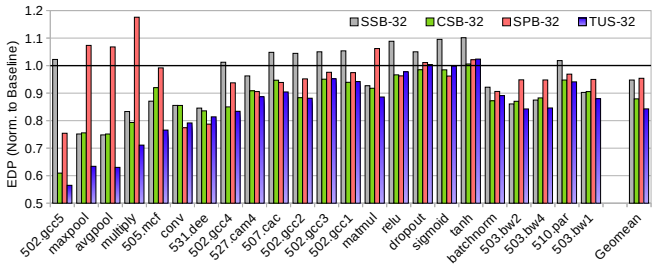

Fig. 15. Normalized EDP to 32-entry SB for single-threaded SB-bound applications. Lower is better

Store-wait–free multiprocessors. Store-wait–free multiprocessors [47] proposes SSB which relies on a large FIFO structure that maintains all stores (TSOB). Similarly to TUS, this approach does not require write permissions to hold blocks that exit the SB in L1D. The key benefit of SSB is the removal of associative search in the large TSOB queue, as store-to-load forwarding is carried out by the L1D. However, SSB drains the TSOB by performing the writes in the shared cache store by store (without allowing coalescing) and in order, which increases energy consumption. Furthermore, it requires extra logic to iterate over the TSOB to perform invalidations. Similarly, Akkary et al. [2] proposed a two-level SB design, with a filter mechanism to reduce snoops in the L2 SB.

InvisiFence [11] uses post-retirement speculation to reduce the performance penalty presented in memory ordering in consistency models. As a speculative mechanism, InvisiFence requires rolling back in case of consistency violations. Violations are detected using read and write sets as in hardware transactional memory and entail post-retirement register checkpointing and a flusheable SB. TUS, on the other hand, only relies on simpler in-window speculation and is able to coalesce stores and reduce SB stalls with a non-speculative solution. Hence, TUS does not require SB/L1 flashing extensions nor post-retirement register checkpointing. In this work, we limited our evaluation to in-window speculation techniques.

VIII. Conclusions

We have presented a mechanism, *Temporarily Unauthorized Stores (TUS)*, to write to the first-level cache without requiring permission, thus draining the store buffer faster. Cache lines written without permission remain invisible to the coherence protocol (i.e., they are unauthorized). TUS enables store coalescing of temporal stores writing to non-consecutive cache lines using the write combining buffers, initially designed for non-temporal stores. Cache lines involved in coalescing operations are made visible to the coherence protocol atomically. This process is performed non-speculatively thanks to the use of a predetermined global order in acquiring and blocking write permissions and the use of *not visible* bits at the first-level cache, that are reset in bulk when making the atomic group of cache lines visible to the coherence protocol.

Our evaluation using gem5 demonstrates that, compared to our baseline processor, TUS improves performance by 3.2% (3.5%) on average for SB-bound SPEC CPU2017 and Tensorflow (Parsec) benchmarks, while reducing the EDP of the processor by 6.4% (5.1%) on average. TUS outperforms all state-of-the-art approaches because it avoids both SB-stalls on long-latency L1D misses *and* supports store coalescing. When configured with a 32-entry SB, TUS yields an average performance improvement of 2% compared to our 114-entry SB baseline while reducing SB energy per search by a factor of 2×, SB area by 21%, and store-to-load forwarding latency from 5 to 3 cycles.

References

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

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: An efficient, scalable alternative to reorder buffers," *IEEE Micro*, vol. 23, no. 6, pp. 11–19, 2003.

[3] R. Alves, A. Ros, D. Black-Schaffer, and S. Kaxiras, "Filter caching for free: The untapped potential of the store buffer," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 436–448.

[4] AMD, "Amd64 architecture programmer's manual, volumes 1-5," https://www.amd.com/en/support/tech-docs/amd64-architecture-programmers-manual-volumes-1-5, 2023.

[5] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 399–411.

[6] V. Balaji and B. Lucia:, "Improving locality of irregular updates with hardware assisted propagation blocking," in *28th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Apr. 2022, pp. 543–557.

[7] R. Bera, A. V. Nori, , O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *52nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2019.

[8] E. Bhatia, G. Chacon, S. H. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 1–13.

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

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

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

[12] J. M. Cebrian, S. Kaxiras, and A. Ros, "Boosting store buffer efficiency with store-prefetch bursts," in *53rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2020, pp. 568–580.

[13] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: Bulk enforcement of sequential consistency," in *34th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2007, pp. 278–289.

[14] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Computing Surveys (CSUR)*, vol. 3, no. 2, pp. 67–78, Jun. 1971.

[15] A. Fog, "Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns," 2018, Available at http://www.agner.org/optimize/instruction_tables.pdf.

[16] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai, "Scalable load and store processing in latency-tolerant processors," *IEEE Micro*, vol. 26, no. 1, pp. 30–39, Jan 2006.

[17] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, X. Wen, R. Ren, C. Zheng, X. He, H. Ye, H. Tang, Z. Cao, S. Zhang, and J. Dai, "Bigdatabench: A scalable and unified big data and ai benchmark suite," 2018.

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

[19] E. J. Gómez-Hernández, J. M. Cebrian, J. R. T. Gil, S. Kaxiras, and A. Ros, "Efficient, distributed, and non-speculative multi-address atomic operations," in *54th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2021, pp. 337–349.

[20] B. Gottschall, L. Eeckhout, and M. Jahre, "TEA: Time-Proportional Event Analysis," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2023, pp. 1–13.

[21] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun, "Programming with transactional coherence and consistency (TCC)," in *11th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2004, pp. 1–13.

[22] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *20st Int'l Symp. on Computer Architecture (ISCA)*, May 1993, pp. 289–300.

[23] IBM, "Power9 processor user's manual," https://openpowerfoundation.org/resources/ibmpower9usermanual/, 2019.

[24] Intel, "Intel® 64 and ia-32 architectures optimization reference manual," www.intel.com, 2019.

[25] ——, "Intel® 64 and ia-32 architectures optimization reference manual," https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual.html, 2023.

[26] ——, "Intel® 64 and ia-32 architectures software developer's manual volume 1: Basic architecture," https://cdrdv2.intel.com/v1/dl/getContent/671200, 2023.

[27] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti, "Friendly fire: Understanding the effects of multiprocessor prefetches," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2006, pp. 177–188.

[28] J. Kim, P. V. Gratz, and A. L. N. Reddy, "Lookahead prefetching with signature path," in *2nd Data Prefetching Championship*, Jun. 2015.

[29] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *49th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2016, pp. 60:1–60:12.

[30] J. R. Larus and R. Rajwar, *Transactional memory*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2007.

[31] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[32] P. Michaud, "Best-offset hardware prefetching," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 469–480.

[33] D. Molka, R. Schöne, D. Hackenberg, and W. E. Nagel, "Detecting memory-boundedness with hardware performance counters," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 27–38.

[34] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibánez, V. Viñals-Yúfera, and A. Ros, "Berti: an Accurate Local-Delta Data Prefetcher," in *55thIEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*. IEEE, 2022, pp. 975–991.

[35] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 118–131.

[36] S. Patel, R. Kalayappan, I. Mahajan, and S. R. Sarangi, "A hardware implementation of the mcas synchronization primitive," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2017, pp. 918—921.

[37] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. fei Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 626–637.

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

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

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

[41] P. Shamis, "Understanding write combining on arm," https://community.arm.com/arm-research/m/resources/1012, 2020.

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

[43] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017.

[44] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek, "Multiple reservations and the oklahoma update," *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 4, pp. 58–71, Nov. 1993.

[45] Thin-Fong Tsuei and W. Yamamoto, "Queuing simulation model for multiprocessor systems," *IEEE Computer*, vol. 36, no. 2, pp. 58–64, Feb 2003.

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

[47] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *34th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2007, pp. 266–277.

[48] S. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *21st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2015, pp. 577–589.

[49] A. Yasin, "A top-down method for performance analysis and counters architecture," *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pp. 35–44, 2014.