Non-Speculative Store Coalescing in Total Store Order

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

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

***Abstract*—We present a non-speculative solution for a coalescing store buffer in total store order (TSO) consistency. Coalescing violates TSO with respect to both conflicting loads and conflicting stores, if partial state is exposed to the memory system. Proposed solutions for coalescing in TSO resort to speculation-and-rollback or centralized arbitration to guarantee atomicity for the set of stores whose order is affected by coalescing. These solutions can suffer from scalability, complexity, resource-conflict deadlock, and livelock problems. A non-speculative solution that writes out coalesced cachelines, one at a time, over a typical directory-based MESI coherence layer, has the potential to transcend these problems if it can guarantee absence of deadlock in a practical way.**

There are two major problems for a non-speculative coalescing store buffer: i) how to present to the memory system a group of coalesced writes as atomic, and ii) how to not deadlock while attempting to do so. For this, we introduce a new lexicographical order. Relying on this order, conflicting atomic groups of coalesced writes can be individually performed per cache block, without speculation, rollback, or replay, and without deadlock or livelock, yet appear atomic to conflicting parties and preserve TSO. One of our major contributions is to show that lexicographical orders based on a small part of the physical address (sub-address order) are deadlock-free throughout the system when taking into account resource-conflict deadlocks. Our approach exceeds the performance and energy benefits of two baseline TSO store buffers and matches the coalescing (and energy savings) of a release-consistency store buffer, at comparable cost.

***Keywords*-store coalescing; memory consistency; store buffer; deadlock free; lexicographical order.**

I. Introduction

Store buffers are indispensable for performance allowing the immediate retirement of store instructions from the pipeline and handling long-latency writes off of the critical path of execution.

Various consistency models accommodate store buffers in different ways. Sequential consistency (SC) [1], which demands strict memory ordering, does not easily accommodate a store buffer except with extensive speculation [2]. In contrast to SC, total store order (TSO), e.g., SPARC TSO [3], TSOx86 [4], relaxes ordering from stores to subsequent loads (relaxes the store→load order, but maintains load→load, store→store, and load→store) *just to accommodate* store buffers. Weak memory models such as release consistency (RC) do an even better job of accommodating store buffers by relaxing all orders except across synchronization. In particular, relaxing the store→store order in RC allows store buffers to *coalesce* stores.

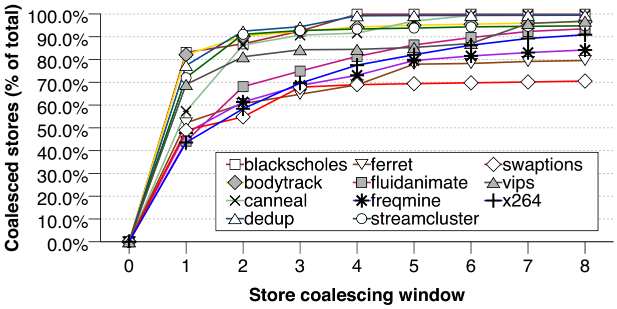


Figure 1: Store coalescing potential

Store→store reordering and coalescing. One of the important benefits of relaxing the store→store order is that we can coalesce *non-consecutive* store operations to the same cache block. Coalescing alleviates store buffer capacity pressure and reduces the number of writes in the L1.

There is ample opportunity for coalescing due to the spatial locality and the burstiness of stores. Figure 1 shows the cumulative coalescing as a function of the number of store operations in a window that allows unrestricted coalescing of stores. With just the 8 most recent stores in the window, PARSEC applications achieve a significant portion of their potential for coalescing.

However, the benefits of coalescing are not easily available in TSO. Coalescing irreversibly changes the order of stores and violates TSO if we allow *partial* state to become visible to the memory system. In Section II-C1 we show how coalescing violates TSO with respect to conflicting loads and in Section II-C2 we show the same with respect to conflicting stores.

In TSO, the store order must *appear* to be maintained even when coalescing is taking place. *Out-of-core speculation* can provide this illusion, but this may be impractical in high-efficiency designs that avoid speculation outside the core, or when invasive changes to the memory hierarchy are not an option. Since these situations are common in industry, a non-speculative solution is needed.

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00028

Coalescing means atomicity. If coalescing is irreversible,[1] the solution is to make the group of stores (whose order is affected by coalescing) atomic. If we can guarantee atomicity with respect to external loads and stores, the order of the stores inside the atomic group does not matter —TSO is preserved.

- Atomicity with respect to loads: Stores appear as one atomic group with respect to conflicting loads if the conflicting loads are ordered either before or entirely after the whole atomic group. This can be accomplished simply by *delaying* a conflicting load until the writing of the whole group completes [2], [5], [6].
- Atomicity with respect to stores: The group of stores appears atomic with respect to a conflicting store (which may be part of another atomic group). This requirement harbors a deadlock (Section II-C2) and needs to be addressed with a deadlock-avoidance mechanism.

Mechanisms for atomic group writes. The ability to atomically write a *group of cachelines* is not trivial as it involves a number of possible deadlocks throughout the memory hierarchy. Two general approaches have been proposed to provide atomicity for a group write:

1. By mutual exclusion with respect to a centralized resource. For example, Transactional Consistence and Coherence (TCC) uses *broadcast* in the network as its "centralized resource:" transactions present their writes atomically to the rest of the system by broadcasting all writes in a single packet to all nodes [7]; In BulkSC instruction "chunks" yield control to a centralized arbiter (or a hierarchy of arbiters) that tests their *read and write signatures* for conflict, and subsequently grants them permission to atomically perform their writes [8] —mutual exclusion is managed by the arbiter.
2. As a generalization of load-linked/store-conditional to multiple addresses, i.e., speculate absence of conflict and roll-back if a conflict is detected [9], [10], [5].

Approaches based on a centralized resource have scalability and contention problems. For example, TCC results in dense write traffic and requires increased network bandwidth to work well [7] and BulkSC increases latency with round trips to the centralized arbiter [8]. Speculate-and-rollback approaches will waste work upon a conflict and are susceptible to livelock or starvation. Additional mechanisms (e.g., aging) must be defined to protect them against these situations [10]. There has never been (as far as we know) a distributed, non-speculative approach to atomic group write that would solve all these issues without any deadlock.

A new perspective. The problem of writing atomically a number of cachelines is similar to the problem of acquiring and holding a number of locks in parallel programming. One of the simplest examples for deadlock is that of two threads trying to acquire two locks but in opposite order: the threads can deadlock. The solution is to always acquire locks in the same order. Similarly, two store buffers trying to atomically perform their writes to two (or more) cachelines can deadlock if the cachelines are written in the opposite order by different cores. Typically, the order of their writes is derived from the program order of the stores. The key idea of this paper is that, since we cannot affect program order to avoid deadlocks, we change to a new non-deadlocking order.

A new solution with lexicographical order. Similar to parallel programming, we impose a *lexicographical* order, ***lex order*** for short, globally agreed in the system, for the store buffers to write their cachelines. Once an atomic group is formed (due to coalescing), we write its cachelines one-by-one in their lex order. In contrast to any previous approach that is a generalization of the load-linked/store-conditional to multiple cachelines, *we do not try to collectively get and hold all the permissions for the cachelines in the group prior to their write and roll-back if we are unsuccessful*. Instead, each cacheline is written immediately if the corresponding write permission is already held (or can be acquired). This means that a conflicting atomic group can *steal* the permission for a cacheline that has not been written yet. In that case, the winning store buffer (the one that gets the permission first) takes priority to finish its writes before the losing store buffer. Throughout the atomic group write and until it is finished, a store buffer *holds on* to the permissions *after* writing a cacheline (by locking it in the L1). This serves a dual purpose: it provides atomicity among conflicting groups and delays any conflicting load from seeing partial group writes. Because of the lex order it is guaranteed that the two conflicting groups will appear atomic with respect to each other without getting into deadlock.

Contributions. *We introduce a lexicographical order for non-speculative atomic group writes*. An obvious lex order is the (physical) address of the cacheline itself but one of our main contributions is to show how *different* lex orders, and in particular *address subsets* can be effectively used to eliminate deadlocks in all of the structures (e.g., private caches, shared caches, directory) that are involved in an atomic group write. The implications of our approach are numerous. Our distributed deadlock- and livelock-free solution can:

- *non-speculatively* (irreversibly) coalesce stores without needing to roll-back *and* atomically write a number of cachelines without the need of broadcast or centralized arbitration;
- interleave at fine-grain (cacheline by cacheline) writes from conflicting atomic groups and prioritize on *first conflict in lex order*;

[1] In Section VI we discuss approaches where the coalesced state is speculative, e.g., the Scalable Store Buffer (SSB) [5]. Such approaches, however, do not write the coalesced state to the memory system, but still expose to the coherence layer individual stores (one-by-one) in their correct order.

- operate with cacheline transactions over an unmodified directory-based, cache-coherence protocol (e.g., directory-based MESI) *and* seamlessly accommodate permission prefetching;
- be entirely implemented in the store buffer and L1 cache controller with minimal additional cost (1-bit per L1 cacheline);
- and finally, free the TSO store buffer from the need to preserve FIFO store order (program order) *and* allow direct-mapped or set-associative implementations, as a cache.

In this paper, we demonstrate our approach by presenting a non-speculative coalescing store buffer (CSB) for TSO, but we note its wider applicability on techniques where atomic group writes are required.

II. Background

A. The Store Queue and the Store Buffer

Store queue. The store queue (SQ) is one of the main structures needed for managing dynamic out-of-order execution inside the core: it handles speculation and enforces ordering between memory operations (including fences). In this paper, we consider that stores in the SQ are within the dynamic instruction window of the processor and have *not committed* yet.

The contents of the SQ need to be searched very often during execution and it is typically implemented as a content-addressable memory (CAM). Its size is limited and must be kept small because of latency and energy considerations. In this paper, we are not concerned about the mechanics of the SQ but instead focus on the Store Buffer.

Store buffer. At the moment a store commits, it is taken out of the SQ and inserted in the store buffer (SB). In our work, we consider this as the point of no return with respect to speculation. A store that leaves the SQ cannot be undone. Regardless of whether the SQ and the SB are implemented separately or combined into a single structure, in this paper, we consider the SB as the structure that contains *committed stores* and sits in the interface between the core and the memory system.

B. TSO Store Buffers

The store buffer in TSO is responsible for enforcing two policies:

- stores are *performed* in the memory system in *program order*; A store is performed when it exits the store buffer and is written into the cache.
- a load from the same core receives its value from the *youngest (in program order) store on the same address*, if such store exists in the store buffer.

These two policies can be satisfied by a single FIFO queue structure that can be searched associatively. *Committed* stores are placed in it and drained to the memory system first-in-first-out (i.e., in *program order*). Loads, in their critical path, search this structure associatively and select the *latest* entry if more than one match exists. A circular buffer/CAM implementation faces significant challenges because of cost, latency, and energy considerations, which translate in store buffers of limited size. We defer the detailed discussion for store buffer designs to Section IV-A.

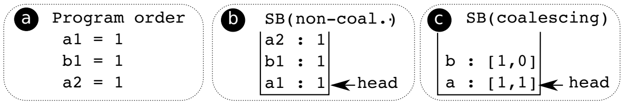


Figure 2: Coalescing violates TSO with respect to loads

Hiding latency. TSO hides the store latency by allowing loads to bypass stores in the store buffer. To make the most out of this, store permissions should be *prefetched* as soon as possible: Store prefetch can start as soon as the store is executed [11] or as soon as it is committed [12]. We prefetch permissions on commit, as implemented in Intel processors [13], to avoid possibly useless prefetching for speculative stores.

Coalescing in TSO. Intel TSO processors do not provide atomicity guarantees when writing more than 8 consecutive bytes [13], which means that coalescing is restricted to this size. However, *consecutive* stores to the *same* cacheline can be coalesced without violating TSO, when the cacheline can be written atomically in the L1. In this work, we assume that this capability exists.

C. How Coalescing Violates TSO

Even with an atomicity guarantee for cacheline writes in the L1, coalescing cannot be done for *non-consecutive* stores in program order, since this would break the total store order. We can show this with respect to conflicting loads (which is well known) but more importantly with respect to conflicting stores. We could not readily identify examples of the latter in the literature, therefore, we introduce a litmus test in Section II-C2.

1) With respect to conflicting loads: Figure 2 shows how coalescing breaks TSO with respect to conflicting loads. Assume that program order is defined by the code example ⓐ. The code snippet writes to two variables a1 and a2 in cacheline a. The difference between a non-coalescing store buffer ⓑ and the coalescing store buffer (CSB) ⓒ is that CSB writes the cacheline a with the value [1,1] whereas the non-coalescing store buffer does this in two stages (first a1, then a2) with the write to b interposed between.

CSB breaks TSO because it is not possible to write the values in program order, if we write one store buffer entry at a time (cacheline granularity). If a is written first, a2 may be seen before b1. Otherwise if b is written first, b1 may be seen before a1.

With respect to loads, one can generalize atomic writing from consecutive stores to the same cacheline to *non-consecutive* stores across multiple cachelines, by simply

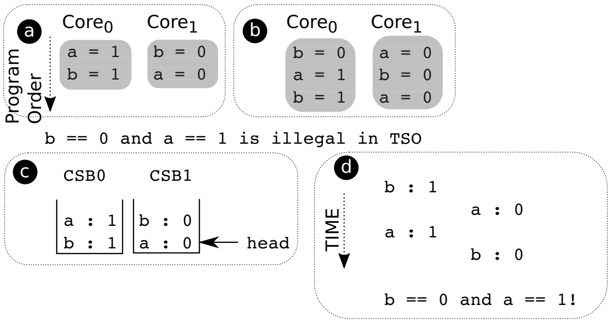


Figure 3: Coalescing violates TSO with respect to stores

delaying the loads until the all the stores, whose order is affected by coalescing, complete [2], [5], [6].

2) With respect to conflicting stores: More importantly, coalescing can violate TSO even if no loads are involved. Figure 3 shows why. Assume the code snippet ⓐ. TSO forbids b == 0 *and* a == 1 after this code is executed (i.e., any future loads that access a and b should not be able to see this combination of values).

Yet, we can easily produce this wrong result with a CSB that writes one store-buffer entry at a time (cacheline granularity). Assume that we expand the code snippet, as shown in ⓑ, to include two more stores (b = 0 in Core$_0$ and a = 0 in Core$_1$) that precede the code in ⓐ. The newly added stores do not change the outcome of the code, neither change the illegal outcome in TSO. The CSBs in Core$_0$ and Core$_1$ will coalesce the writes to b and a respectively as shown in ⓒ.

Assume now that the CSBs in Core$_0$ and in Core$_1$ output their writes in the order shown in ⓓ. The end result is the set of values b == 0 *and* a == 1, which is illegal in TSO. The reason why this happens is because the coalescing of b in Core$_0$, changes the order in which a and b are written by CSB$_0$; analogously, coalescing of a in Core$_1$ changes the order in which a and b are written by CSB$_1$.

The obvious solution is to write all the stores that are *engulfed* by coalescing *atomically* with respect to conflicting stores from other cores. This means that Core$_0$ should write a and b as an atomic group (since a is enclosed by the coalescing of b) and Core$_1$ should also write a and b atomically (since b is enclosed by the coalescing of a).

But note, now, what happens in ⓓ. Assume that Core$_0$ will attempt to get the write permissions for *both* b *and* a before outputting its writes. Similarly Core$_1$ will attempt to get both the write permissions for a and b. Core$_0$ gets the permission for b first and Core$_1$ gets the permission for a first: this leads to **deadlock**. Without a mechanism to guarantee deadlock (and livelock) freedom, coalescing of non-consecutive stores cannot be done for TSO.

III. Lexicographical Order

The deadlock problem of getting the permissions for a *group of stores* has been addressed before in two ways: i) by *mutual exclusion* on a centralized resource for the group write (e.g., broadcasting in the network [7], or obtaining permission for the whole group from an arbiter [8]); or ii) by *speculation-and-rollback* as a generalization of load-linked/store-conditional [9]. As far as we know, there is no known solution to this problem that does not involve a centralized resource, or speculation-and-rollback. In this section, we propose such a new solution. As we argue in the introduction and later on in Section VI, prior solutions are plagued by a number of problems (including vulnerability to livelock, or *resource conflict* deadlocks) and thus are a challenge to implement in real designs. Our solution encompasses a unified approach to all these problems.

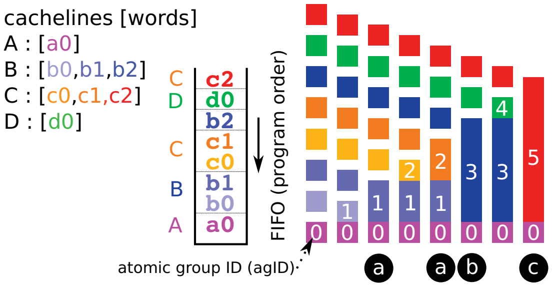


Figure 4: Forming atomic groups

A. Forming Atomic Groups

We first discuss how atomic groups are formed because of coalescing. Figure 4 shows coalescing on a number of cachelines (A to D). The figure shows three cases of coalescing. Case ⓐ shows the simple case where consecutive stores coalesce in the same cacheline. Case ⓑ shows the case where a number of stores are *engulfed* by a pair of coalescing stores (b2 and b0) at the endpoints: they all become part of the same atomic group. Case ⓒ shows the slightly more complicated case of chained engulfment. Here, two pairs of coalesced stores (b2, b0 and c2, c0) are interleaved and merge their atomic groups into one.

The algorithm to form atomic groups is the following: each store is tagged (numbered) with a new *atomic group ID* (*agID*) when entering the CSB. Only as many agIDs as the size of the CSB are needed. If a new store coalesces with a previous store of some agID *the new store forces its agID to all the engulfed atomic groups between itself and the coalescing atomic group, inclusive*. Each atomic group is written in agID order with respect to other atomic groups. Cachelines within an atomic group, however, can be written in any (non-deadlocking) order. The decision on when to start writing an atomic group is an implementation option and is discussed separately in Section IV-B

B. Address Order

In Figure 3 ⓐ, Core$_0$ attempts to write {b, a} at the same time as Core$_1$ attempts to write {a, b}. In each store buffer, the order a and b are written follows the order in which they

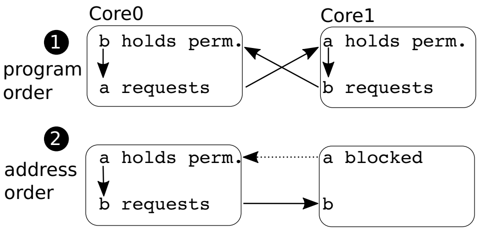


Figure 5: Writing atomic groups in address order

```
Each Coalescing Store Buffer (CSB) performs the following:

1:   foreach atomic-group (oldest to youngest) in the CSB do
2:     left_to_do = number of cachelines in atomic-group
3:     repeat
4:         min=0xFFFFFFFF
           // find the min
5:         foreach cacheline in the atomic-group {
6:          if (not completed)
7:              if (addr <= min) min=addr;
8:         }
           // write it and lock it
9:         {   write min;
               mark min ``completed'';
               lock L1 cacheline min }
10:       left_to_do  --
11:    until left_to_do == 0
12:    unlock all cachelines
13:  end foreach
```

Figure 6: Address order algorithm

were allocated in the store buffers. The allocation, in turn, is the result of the program order in the respective cores. As we have seen, this order creates a cycle (deadlock) shown in Figure 5 ①.

The key observation is that in an atomic group write we do not have to follow any particular program order for the writes. Instead, we can simply chose an order that will not deadlock. In Figure 5 ② we pick the *address order*: we write cachelines in ascending order of their address. In the particular example, CSB_0 writes a first. It will not release a until it finishes its atomic group. This means that CSB_1 will have to wait to write a and since it must write in ascending address order it cannot write (and hold on to) b —which would be the cause of the deadlock. Even if $Core_1$ (CSB_1) has the permissions for b it relinquishes them to CSB_0 upon request.

Figure 6 generalizes the above example for an arbitrary number of store buffers, each containing an arbitrary number of atomic groups, each consisting of an arbitrary set of cachelines.

Each store buffer writes its atomic groups in order (oldest to youngest). The main part of the algorithm (lines 2 to 12) writes an atomic group. Among the cachelines of an atomic group that have not been written yet, the algorithm selects the one with lowest (min) address. The key functionality starts on line 9. The L1 is accessed and if the corresponding cacheline has write permissions, the store buffer writes its updates immediately. We say *updates* because most of the time only part of the cacheline changes in the store buffer: the updated bytes are merged with the cacheline in the L1. If the L1 cacheline does not have the proper permissions (or it is a miss in the cache), a request for the permissions is sent and the store buffer waits for the completion of the request. It will resume when it gets the permissions. When the write succeeds, the cacheline is *locked* in the L1 — using an extra bit per cacheline— and marked as "completed" in the store buffer.

A locked L1 cacheline does not relinquish its permissions to an external write request nor does it allow reads. Instead, it delays both external read and external write requests until the lock is removed.[2]

All the lock bits are reset in bulk when the atomic group write completes all its writes to the L1. Delayed read and write requests are satisfied at this point, before a new atomic group write is started, ensuring forward progress for stalled atomic group writes of remote CSBs.

The correctness of this algorithm can be traced back to the theoretical foundations discussed by Coffman et al. in [15], and even to the solution of Dijkstra's dining philosophers problem [16]. By writing and locking cachelines in ascending address order it is guaranteed that store buffers cannot deadlock by forming a cycle.

Propetries. Our solution exhibits a number of properties that distinguish it from prior work:

- It seamlessly supports *permission prefetching* as early as possible, as this is important for TSO performance [11]. A prefetch to a locked line is NACK'ed and fails. Prefetching is safe, even if it happens in program order. It does not deadlock. In contrast, permissions are *demanded* when the store buffer writes a cacheline in the cache. The algorithm in Figure 6 only blocks for permission *demands* that hit on locked cachelines.
- No attempt is made to lock *beforehand* the permissions for all the cachelines in an atomic group. Instead, lock acquisition is incremental, cacheline-by-cacheline, and only when a cacheline is written. This provides flexibility and scalability. In contrast, some prior approaches aim to lock all permissions at once which necessitates either a centralized authority (e.g., [8]) or roll-back (e.g., [9]).
- On a conflict, already acquired permissions are *not* relinquished. In contrast, speculate-and-rollback approaches give up all permissions and are back to square one (e.g., [9]). The difference here is vulnerability to livelock.
- If two atomic groups have multiple conflicting cachelines, they will conflict *only* on the cacheline with the *minimum common address*. In a sense, we "replace" the set of potential conflicts with a unique (for both sides) point

[2] Note that, there can be only one *blocked* external read or external write request per cacheline, in a system with a blocking directory (e.g., as in the GEMS implementation [14]). Since conflicts can happen only for an ongoing atomic group write, the buffering space for delayed requests is bounded by the size of the atomic group in the store buffer, and thus poses no threat for deadlocks or livelocks due to overflow.

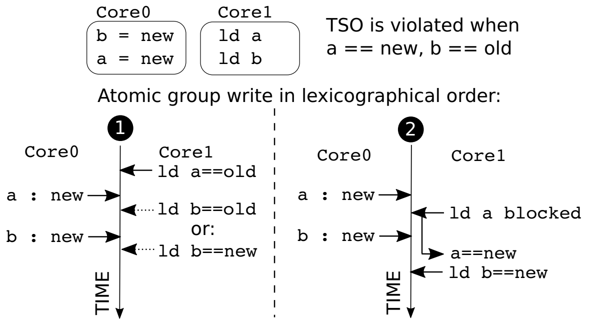


Figure 7: Atomicity with respect to loads with address order

of conflict (*minimum common address*). In contrast, other approaches attempt to map the set of potential conflicts on a new entity (e.g., a signature) and this introduces an additional step in the algorithm: an interaction with some centralized authority. We do not require this *extra* step.

- Our approach solves the long-standing problem of providing atomicity to stores that straddle two cache lines, by creating an atomic group of the two cachelines when such situations are detected.[3]
- Finally, while the naïve algorithm we present in Figure 6 is $O(n!)$ for an atomic group of size n, it is implement in hardware as $O(n)$ (Section IV-A).

To summarize:

Deadlock: Deadlock in getting permissions and locking cachelines (assuming no other resource restrictions for now) is avoided by using *ascending address order* [15], [16].

Livelock: Livelock is avoided because: i) our approach does not roll-back on conflicts; and ii) conflicting requests are delayed and satisfied in order when the writing of the atomic group completes.

C. Atomicity with Respect to Loads

Holding all permissions is necessary to guarantee atomicity for writes (Section II-C2) but also satisfies atomicity with respect to loads (Section II-C1): As we explain in the previous section, a cacheline is locked only at the moment it is written as part of an atomic group. Prior to being locked the cacheline is available for reads. Even if a load is in conflict with one of the stores in an atomic group, *the load is allowed to read the old value if the new value has not been written yet*. This is correct in TSO. As far as the core that has issued the load is concerned, none of the stores in the atomic group are visible, even though some cachelines may already have been written.

However, if a conflicting load bumps on a locked value (i.e., a new value) written by an atomic group, then the conflicting load is stalled until the whole atomic group finishes. This guarantees that the stores in the atomic group cannot be observed out-of-order by a racing core.

Consider the example in Figure 7. The code executes `st b` and `st a` in Core$_0$ and the respective loads in the opposite order in Core$_1$. TSO is violated if `ld a` sees the new value and `ld b` sees the old. Note, however, that lex order *reverses* the order that the stores are performed in the memory system. If `ld a` reads cacheline `a` before it is written (case ①), it gets the old value and it does not matter what value `ld b` sees (either new or old, depending on the interleaving, is fine with TSO). If, on the other hand, `ld a` attempts to read the new value of `a` (case ②) it will be stalled until `b` is also written. Stalling `ld a` delays `ld b` in Core$_1$ on account of TSO's load→load order.[4] The end result is that if `ld a` is bound to see the new value of `a`, its completion after the writing of the atomic group ensures that `ld b` will also see the new value of `b`, as required by TSO. To summarize:

Deadlock: Delaying loads (until the writing of stores completes) does not cause a deadlock as long as it is guaranteed that the writing of stores will complete. Stores can only be delayed by conflicting stores. But, as we show in the previous section, write conflicts among atomic groups cannot deadlock and prevent the completion of stores. From this perspective, both loads and stores cannot be prevented from completing.

Livelock: Livelock is avoided because conflicting loads are just delayed for the duration of the atomic group write. Stalled read requests are satisfied (in order) at the end of the atomic group write and before a new one begins.

D. Resource-Conflict Deadlocks

So far, we have seen that using the address as the lex order solves the write-and-lock deadlock for atomic group writes, assuming no other resource restrictions. The requirement to hold on to all locked cachelines until we finish an atomic group write, however, introduces *resource-conflict* deadlocks in the various points in a real system. Such deadlocks have been sometimes ignored in prior work (e.g., [9]).

The two types of deadlocks that can appear in various structures in the memory hierarchy are due to:

Intra-group conflict in private structures: elements of the same atomic group cannot be accommodated together. The example here is the cache associativity deadlock. We must be able to write and hold on to all permissions of the cachelines in an atomic group simultaneously. If several elements of an atomic group map to the same cache set and their number exceeds the associativity of the set, the algorithm in Figure 6 deadlocks. In general, a deadlock avoidance mechanism cannot simply rely on increasing associativity (e.g., with the addition of a fully associative victim cache) as this only makes the deadlock more unlikely, but not impossible.

[3] This solution holds even when a store straddles the boundary of a sub-address lex order. Sub-address order is discussed in Section III-E.

[4] If `ld b` executes speculatively, before `ld a` completes, it will be squashed (to preserve TSO) when `b` is written by Core$_0$ and eventually see the new value.

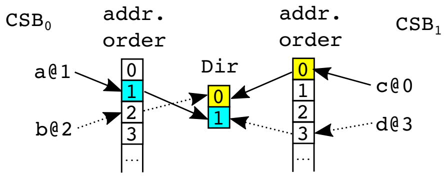


Figure 8: Directory-eviction deadlock

Note, here, that the resource conflict *does not concern* reads (loads), as the inability to evict means that read will be delayed until the atomic-group write completes—or the read will be performed as *uncacheable* [17].

Inter-group conflict in shared structures: elements of different atomic groups deadlock contenting for a *shared* resource. The example here is the directory-eviction deadlock (or, analogously, an LLC-eviction deadlock but for simplicity we will use the directory to stand in for the general case of a shared structure). We must be able to hold all the permissions of an atomic group simultaneously but some other atomic group interferes at the directory. If two atomic groups need to allocate new directory entries but they are prevented by eachother, they deadlock. This happens when each atomic group already occupies the contested directory entry needed by the other atomic group, and *will not let go*. A directory eviction must invalidate the writer. But if the writer has locked its cacheline it will not respond until the end of the atomic group write in progress.

Figure 8 shows a very simple example of this situation. CSB$_0$ and CSB$_1$ attempt to write their atomic groups: {a, b} and {c, d} respectively. The atomic groups have no cachelines in common and should not conflict in their writes. CSB$_0$ first writes a with address 1 and then attempts to write b with address 2. CSB$_1$ first writes c with address 0 and then attempts d with address 3. Moreover, assume that we have a direct-mapped directory with just two entries (i.e., an index of one bit). The topmost entry of the directory (index 0) is occupied by c and the bottom entry (index 1) by a. Because a and c are locked in the respective caches the directory entries cannot be evicted. This is a deadlock. The CSBs cannot write b and d, and cannot finish their atomic writes.

The reason for this deadlock is that the order in which two (or more) atomic groups *allocate* entries in the directory deadlocks. In other words, the lex order maps in such a way to the directory structure so that elements that would be ordered in lex order contend for the same positions but in a deadlocking order.

This resource conflict does not concern reads (loads) even when they can be blocked by an atomic group write. This is because if a read needs to allocate a directory entry, the write of that cacheline in the atomic group has not happened yet. The read gets the old value from memory, regardless of whether it can allocate a directory entry or not. On the other hand, if a read *blocks* waiting for an atomic group, then it is using a directory entry already allocated by the write to the cacheline by the atomic group, and thus is covered by the discussion above.

E. Sub-Address Order

One of the key contributions of this work is to show that there is a system-wide lex order that is *deadlock-free* for both write conflicts and resource conflicts for all the cache structures in the system, private (e.g., L1, L2) or shared (e.g., LLC, directory).

Direct-mapped private structures. Let us take the simplest case and imagine that we have a direct-mapped cache. Obviously two elements of an atomic group that map to the same position in the direct-mapped array, cannot be accommodated. We solve this by demanding that two distinct items in an atomic group cannot map to the same position. This means that the *largest size* lex order that we can use to order the items in an atomic group is simply the *index of the direct-mapped array*, so that:

$$lex\ rank = addr_{cacheline}\ \%\ index$$

Correctness of the algorithm in Figure 6 with sub-address (index) lex order can be understood using the theoretical framework of [15]. In this framework, a number of processors contend for access to multiple shared resources. Sub-address lex order corresponds to dividing contended resources in classes and then ordering these classes. As long as only a single item *per class* is requested by the processors, deadlock is avoided by the ordering of classes [15]. In our case, the contended resources are cacheline addresses. They are divided into classes where each class contains all the addresses *of the same index*. Finally, we order the classes by the index. The implication is that an atomic group cannot accommodate distinct cachelines with the same index (*rank*) in lex order because we cannot order such cachelines.

Forming atomic groups in sub-address order. Each time the store buffer gets a new cacheline whose index clashes with the index of *another* cacheline already in the store buffer —*a lexicographical order conflict*— we must start a new atomic group and stop coalescing. Coalescing is not allowed between the new cacheline and any cacheline in the "clashing" atomic group. Attempting otherwise would bring the two cachelines into the same atomic group. Transitively, the new cacheline is not allowed to coalesce with *any* atomic group that is *older* than the clashing atomic group. The clashing (*rank-conflicting*) atomic group in the store buffer turns into a *coalescing barrier* and prevents any further coalescing for itself and all older atomic groups. This is a new condition that enters into force with respect to the algorithm for forming atomic groups that we describe in Section III-A.

Set-associative private structures. It is now straightforward to expand our approach from a direct-mapped to a set-associative cache. A set can only accommodate up to

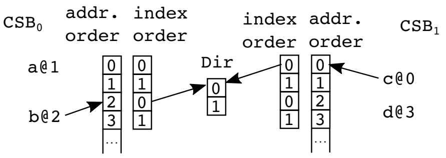


Figure 9: Directory-eviction with index order

$assoc$ items, where $assoc$ is its associativity. The number of candidate cachelines that can occupy these $assoc$ positions in a set is of course much larger. If we want to restrict an atomic group so it does not contain more than $assoc$ items that can map to the same set, the lex order that we must use can be at most $assoc \times index$. In other words, we add to the lex order $log_2\ assoc$ bits from the address[5] for a total of $log_2\ assoc + log_2\ index$, so that:

$$lex\ rank = addr_{cacheline}\ \%\ (index \times assoc)$$

For example, in a 4-way set-associative cache the lex order is 4 times larger than the index. This guarantees that at most 4 elements from the same atomic group can map to the same set, eliminating any chance for a conflict.

Note, that in no way we restrict what can possibly go in a cache set or in the store buffer. The cache and the store buffer work as before. We only restrict the *formation of atomic groups* in the store buffer and consequently the order in which we write the contents of the store buffer to the cache.

Shared structures. Figure 9 shows how *index order* solves the directory-eviction deadlock of Figure 8. We use the directory's one-bit index ($index_{dir}$) as the lex order for the atomic group writes, instead of the address order that was used in Figure 8. Writes from different atomic groups conflict on the allocation of a directory entry only if they are of the same rank in lex order. We call this **lexicographical eviction**. For example, in Figure 9, only one of b or c, which have a rank of 0, can be the first to allocate the directory entry 0.

Assume now that b is first in the directory entry 0. If c tries to evict b, it must send an eviction invalidation to the cached copy of b. This eviction invalidation reaches Core$_0$ and will be considered by the algorithm in Figure 6 in the same way as any other *write conflict*, i.e., an invalidation due to another core writing the same address b. If b is locked in the L1 of Core$_0$, the invalidation has to wait; otherwise, the cacheline is invalidated and the directory entry can be replaced by c. The winning store buffer is the one that first gets the directory entry *and* manages to write-and-lock its cacheline. The losing store buffer has to wait for the eviction until the winning store buffer finishes writing its atomic group.

Table I: Lexicographical order in an example system

Structure (Size)	Index ($\log_2$)	Assoc ($\log_2$)	Index×Assoc ($\log_2$)
L1 (32KB)	64 (6)	8 (3)	512 (9)
L2 (128KB)	256 (8)	8 (3)	2048 (11)
LLC (8MB)	16384 (14)	8 (3)	131072 (17)
Dir (32K-entries)	4096 (12)	8 (3)	32768 (15)

Lexicographical eviction in set-associative shared structures. As before, we can extend our approach from a direct-mapped to a set-associative directory by using a sub-address lex order. The *largest size* lex order that guarantees no deadlock is: $index_{dir} \times assoc_{dir}$. In this lex order, a single atomic group can claim up to $assoc_{dir}$ entries in a set. Two or more atomic groups can collectively fit up to $assoc_{dir}$ entries, *non-conflicting in lex order*. To deadlock, one atomic group must try to fit one more entry (which would require an eviction). But the lex order guarantees that this new entry must conflict in rank with an entry already present in the set (pigeonhole principle). The new entry must evict the old with the same rank (lexicographical eviction) —not any other replacement victim, e.g., not necessarily the LRU. As we explained above for the direct-mapped directory, this eviction will be correctly handled by the algorithm in Figure 6.

However, this assumes that *all* eviction in the directory (including eviction by reads) must abide by lex order: *evict the same rank if present*. This overrules the normal replacement policy. Under this assumption, a lex order of $index_{dir} \times assoc_{dir}$ renders the directory *direct-mapped* no matter what its physical organization and its normal replacement policy may be. Fortunately, this does not happen if the size of the lex order is *smaller* than or *equal* to $index_{dir} \times assoc_{dir}$. As we argue below, this holds for any reasonable system.

Putting it all together. Consider a system that consists of caches (private L1 and L2 per core) and shared structures (LLC and directory). A write may need to visit all these structures. What is the *largest, system-wide* lex order that guarantees deadlock-free passage through all these structures? The answer is the minimum $index_i \times assoc_i$ of all the structures $i = 0..n$ in the system, so that:

$$lex\ rank = addr_{cacheline}\ \%\ min(index_i \times assoc_i)$$

As an example, let us consider the configuration that is used in our evaluation (Section V), shown in Table I. The lex order is determined by the L1 and is 2^9.

The size of this lex order is 16 times smaller than the index of the LLC and four times smaller than the index of the directory. In these two structures, any item that maps to the same set has the same lexicographical rank. Therefore, replacement in the shared structures is free to follow any desired replacement policy. The shared structures are not turned into direct-mapped on account of the lexicographical eviction.

[5] Obviously, if associativity is *not* a power-of-two we select the next lower power-of-two.

In general, we can be assured that in any reasonable system the shared structures will not be limited by lexicographical eviction, as the lex order is defined by the minimum-size private structure, which is the L1. If the lex order is at most as large as the index of a shared structure, the structure's associativity is not constrained. It is only when the lex order is larger than the index (and items of different ranks map to the same set), the associativity is artificially constrained because of lexicographical eviction. This is unlikely in practice because:

- The directory and the LLC are a function of system size (number of cores) but the lex order is constant, fixed by the size of a single (smallest) private cache. Thus, we can expect the directory and LLC to grow with the number of L1 caches, while lex order remains tied to the size of a single L1.
- The directory covers at least the total capacity of the largest private caches and is often over-provisioned 2× (or more). The L2, in turn, is typically at least twice as large as the L1. Thus, even in the smallest 2-core system (with an L2 twice as big as the L1, and 2× directory over-provisioning) we can expect that a directory associativity of eight ($2 \times 2 \times 2$) is not constrained by lexicographical eviction.

Finally, note that MSHRs do not affect lex order. We assume that one MSHR is always reserved for writes which happen in lex order. In other words, loads (that can block) cannot hog all MSHRs and prevent stores from completing.

IV. Implementation Details

A. Store Buffer Organization

In general, non-coalescing TSO store buffers are limited by the need to drain the store values in program order into the memory system. To satisfy this requirement, an *age-indexed organization* can be used, for example, a FIFO circular buffer with a head and tail pointer, where store-value entries are inserted in the tail and extracted from the head. With an age-indexed organization, allowing loads in their critical path to *quickly* search the store buffer, necessitates a CAM functionality on address.

Fully-associative LSB. A line-coalescing store buffer (LSB) design simply replaces each individual store entry with a cacheline. Consecutive stores can coalesce only in the *last-in* cacheline. Figure 10 ⓐ shows an 8-entry line-coalescing SB as a circular buffer that also doubles as a single 8-way associative set (CAM). Time and energy constraints come from the following operations:

- Store-insert: Stores insert values in a *direct-mapped* fashion (in the entry pointed by the tail pointer).
- Load-search: Loads search the buffer *associatively* as an 8-way set, looking for the *youngest* match, i.e., the associative search is *prioritized* by the position in the circular list.

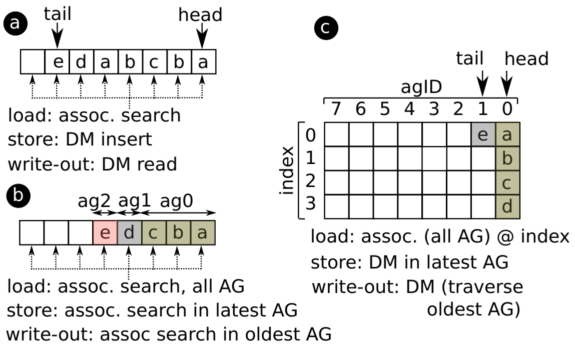


Figure 10: Line-coalescing LSB, fully-associative CSB, and lexicographical CSB

- Write-out: The entry at the head of the SB (*direct-mapped*) is written out.

Fully-associative CSB organization. A *coalescing* store buffer (CSB) changes the requirements for the organization. What matters in our case is: i) the age order of the atomic groups, and ii) the lex order of the addresses within an atomic group. A first simple organization is a fully-associative organization (CAM) similar to the LSB and is shown in Figure 10 ⓑ. In this organization an atomic group spreads across the associativity dimension (e.g., `ag0` with `a`, `b`, and `c`). As with the LSB, each entry can be associatively matched on address. An extra field, the atomic group ID (agID), requiring an extra $log_2(SB_{size})$ bits per entry (see Section III-A), is used for prioritization.

- Store-insert: Stores search for the youngest same-address entry for coalescing. If no match is found, or if a lexicographical conflict is detected in-between, the store is inserted (*direct-mapped*) at the tail pointer. Otherwise coalescing takes place on the matched entry and tagging of the store buffer entries with new agIDs is performed according to Sections III-A and III-E, off the critical path.
- Load-search: Loads search the buffer *associatively* as an 8-way set, looking for the *youngest* match; the search is also *prioritized* by the position in the circular list.
- Write-out: Select the entries of the oldest atomic group (agID); sort on sub-address lex order —sub-field of the address tag— using a priority encoder tree or other similar techniques [18].

This organization is well fitted for small (e.g., 8-entry) CSBs and provides significant coalescing. It is slightly more expensive in time and energy for the store-insert and the write-out operations (which are not time-critical) but it is the same as the LSB for the load-search. The increase in time and energy over the corresponding LSB operations is negligible with respect to the resulting difference in writes to the L1.

Lexicographical CSB organization. Figure 1 shows that, for the workloads we examine, small to medium-sized store buffers (e.g., 8 to 32 entries) can capture the majority of the coalescing potential. However, if need be, for workloads that are write-intensive and where fast store propagation does not play a major role in performance (e.g., see the workloads in [5]) there is an alternative organization based on lex order that makes the CSB efficient and *scalable*. The key notion here is that while an LSB cannot be scaled except in associativity (which is the expensive dimension) a CSB can be scaled in the number of sets (which is the *indexed* dimension). Figure 10 ⓒ shows the lexicographical organization.

In this organization each associative way becomes a separate atomic group. The policy of forming atomic groups changes in this case, from the one described in Section III-A. By default we now pile up as many stores as we can on the same atomic group (e.g., `a` to `d` in `ag0`). Each way (atomic group) is a *direct-mapped buffer already sorted in lex order*. We are forced to start on a new atomic group only on *lexicographical conflict* (e.g., `e` with `a`). In this model:

- Store-insert: *direct-mapped*, current atomic group (way), indexed by lex order.
- Load-search: *associative* on the set indexed by lex order.
- Write-out: *direct-mapped* based on a traversal of a single (oldest) associative way.

While this organization addresses the need for large, scalable store buffers, when the need arises, we do not examine it further since, for our workloads, we do not achieve significant benefits by increasing the size of the store buffer (a large portion of the potential for coalescing in our workload is covered with few entries, see Figure 1).

B. Policies to Modulate Coalescing

Policies for modulating coalescing have been studied previously [19], [20]. An occupancy-based approach allows some time for coalescing but starts draining the store buffer when a high water mark is reached [19]. In our case, the high water mark puts a limit on how large an atomic group can grow before writing it out. However, previous studies concern single-threaded workloads where slow store propagation in the memory system is not an issue. For a parallel workload one must strike a fine balance between delaying stores for coalescing and delaying other threads from seeing new value [6]. In our case, we use an occupancy-based policy with a high water mark but enhanced with a time-out (to ensure progress if the high water mark is not reached).

V. Evaluation

We simulate a multicore processor consisting of 8 out-of-order cores. Our simulation infrastructure is based on the cycle-accurate GEMS simulator [14] for multicore systems, which offers a timing model of the memory hierarchy and the cache coherence protocol. A detailed x86-like in-house out-of-order processor model driven by a Sniper [21] front-end has been incorporated into GEMS. Our processor model implements a fully pipelined (both reads and writes) L1 cache with next-line prefetching. The interconnect is modeled with GARNET [22]. The architectural details of the simulated system are displayed in Table II. We run the PARSEC 3.0 applications [23], with *simsmall* (freqmine, streamcluster, swaptions, and vips) and *simmedium* (blackscholes, bodytrack, canneal, dedup, ferret, fluidanimate, and x264) inputs, and present results for their region of interest.

Table II: System configuration

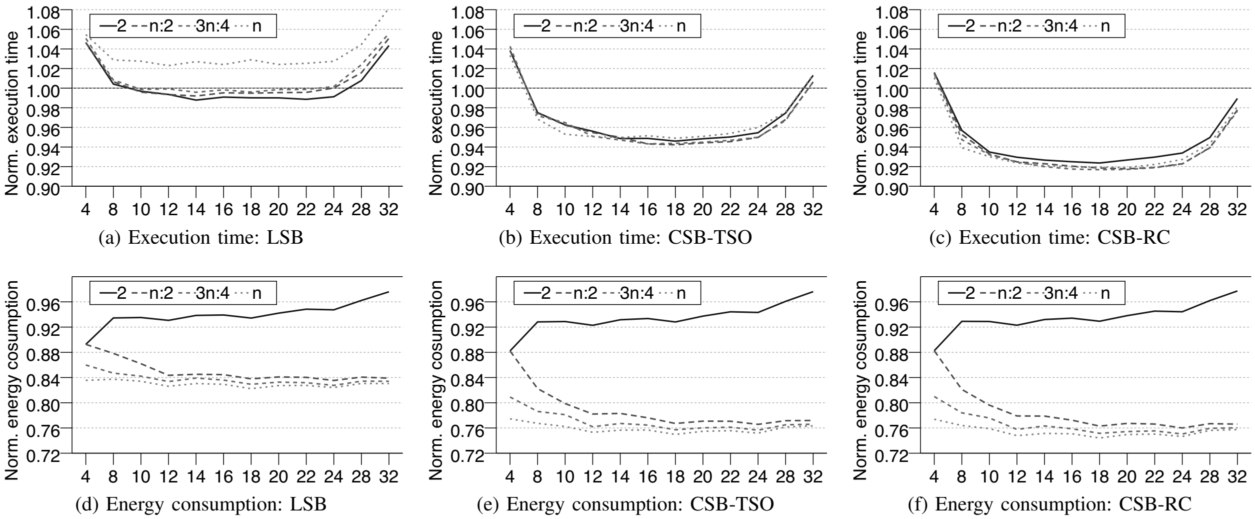
Processor	
Issue / Commit width	4 instructions
Instruction queue	60 entries
Reorder buffer	192 entries
Load queue	72 entries
Store queue + store buffer	42 entries
Memory	
Private L1 I&D caches	32KB, 8 ways, 4 hit cycles, pipelined
Private L2 cache	128KB, 8 ways, 12 hit cycles
Shared L3 cache	1MB per bank, 8 ways, 35 hit cycles
Directory (8 banks)	512 sets, 8 ways (200% coverage)
Memory access time	160 cycles
Network	
Topology	Fully connected
Data / Control msg size	5 / 1 flits
Switch-to-switch time	6 cycles

Four SQ/SB configurations (some described in Section IV-A) are modeled. A non-coalescing, unified SQ/SB (NSB), similar to the one implemented in Intel processors [13], is the baseline on which we normalize our results. The remaining configurations employ separate structures to allow coalescing. LSB (line-based) coalesces in the last cacheline if there is a match. These store buffers do not violate TSO. CSB-TSO (based on lexicographical order) and CSB-RC (coalescing, release consistency) allow coalescing to non-consecutive lines. CSB-RC, however, performs the writes in any order thus relaxing TSO. For every configuration, the total number of SQ+SB entries is 42. This restriction is imposed by the requirement to perform an associative search of *both* structures on every load to ensure proper store forwarding. The SQ and the unified SQ/SB of our baseline have 32-byte entries and the coalescing store buffers in LSB, CSB-TSO, and CSB-RC have 64-byte entries. We obtain energy consumption for the SQ, SB, and L1 cache with CACTI-P [24] for a 22nm process technology.

A. Sensitivity analysis

A key advantage of the non-coalescing baseline, with respect to the coalescing solutions, is having a unified SQ/SB with a dynamic partitioning between the SQ part (uncommitted stores) and the SB part (committed stores). This yields a better utilization of the total number of entries available. The coalescing solutions require a fixed partitioning of the SQ and SB, thus, potentially introducing more stalls (i.e., when either the SQ or the SB become full), even if the total number of entries would be enough to avoid a stall.

The sensitivity analysis for execution time and energy consumption (both averages over all benchmarks) with respect to the SB size is shown in the graphs of Figure 11.



(a) Execution time: LSB

(b) Execution time: CSB-TSO

(c) Execution time: CSB-RC

(d) Energy consumption: LSB

(e) Energy consumption: CSB-TSO

(f) Energy consumption: CSB-RC

Figure 11: Sensitivity analysis (execution time and energy consumption) for LSB, CSB-TSO, CSB-RC, normalized to NSB. The *x*-axis is the number of SB entries, where: SB entries + SQ entries = 42. Four policies for draining the SB are plotted per graph.

In all graphs, the *x*-axis represents the number of entries in the SB. On each graph, four policies for the draining of the store buffer are plotted: i) Start draining when a second entry is inserted in the SB (and no more coalescing is possible in the LSB), i.e., a high water mark of 2 entries. ii) Drain with a high water mark of half the entries in the SB (n:2). iii) Drain with a high water mark of three quarters of the entries in the SB (3n:4). iv) Drain when the SB is full (n). For LSB, a high water mark larger than 2 degrades performance (since there are no extra coalescing opportunities). For CSB, larger high water marks are beneficial as they increase coalescing. In the remainder of this evaluation we will employ a high water mark of 2 for LSB and n:2 for CSB.

Overall, for execution time (graphs (a) to (c) in Figure 11) we observe the following: A very small SQ or SB increases the number of processor stalls and therefore execution time. CSB-TSO gains a significant advantage over LSB and approaches the performance of CSB-RC. An SB of 18 entries yields optimal results for separate queue implementations, and we will use this configuration (18-entry SB / 24-entry SQ) for the rest of the evaluation to present the individual benchmark results. Regarding energy consumption, results improve with SB size as more coalescing is taking place, but only up to the point where the coalescing potential of the applications is exhausted.

B. Energy Implications

Coalescing stores. Figure 12 shows the percentage of stores that coalesce in the store buffer. NSB does not perform coalescing. As expected, coalescing non-consecutive stores yields a difference between LSB and CSB. LSB coalesces a moderate number of stores (48%), while CSB-TSO and CSB-RC coalesce around 87% of all stores.

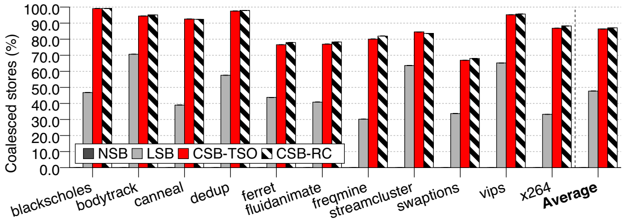


Figure 12: Percentage of coalesced stores

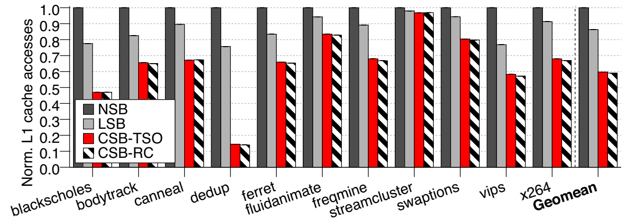


Figure 13: Normalized number of L1 cache accesses

Accesses to L1 cache. Figure 13 shows the normalized number of accesses to the L1 cache. The main effect of store buffer coalescing is to reduce energy-costly L1 writes and replace them with associative store-buffer accesses. A side effect is the filtering of L1 reads, which is explained below.

Energy consumption. Confirmation of the energy savings is given in Figure 15. This graph is based on our CACTI-P modeling of all the SQ and SB operations and L1 accesses and breaks-down the energy consumption in six categories. SQSB_Search accounts for both load searches and store searches (fully associative in CSB-TSO and CSB-RC). L1_Tag corresponds only to misses and prefetches; tag consumption of reads/writes is included in L1_Read/L1_Write.

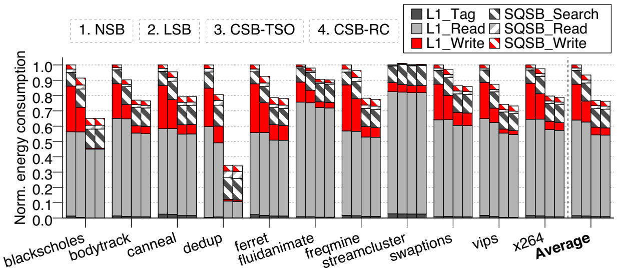


Figure 14: Normalized energy consumption

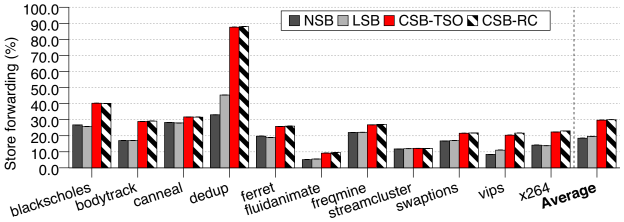


Figure 15: Store forwarding

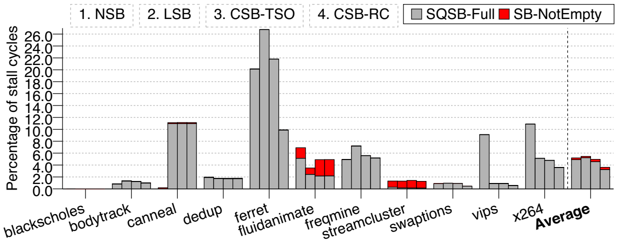


Figure 16: Processor stalls because of the store buffer

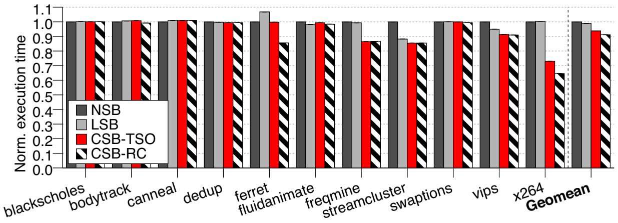


Figure 17: Normalized execution time

The main savings come from the reduction of L1 reads and writes which are transformed into a combination of store buffer operations (search, read, write), depending on the design (Section IV-A). The reduction in L1 writes increases the search operations in the CSB-TSO and CSB-RC. The reduction in L1 reads comes from hits in the store buffer: as we keep more stores in the store buffer, we increase its hit rate, filtering out cache reads. The increase of *store forwarding* through the coalescing store buffers is shown in Figure 14. Overall, coalescing (CSB-TSO and CSB-RC) raises the percentage of loads that hit in the store buffer to 30% (up from 18% and 19% for the NSB and LSB, respectively). In `dedup` we are witnessing an intense store forwarding phenomenon.

On average, CSB is the most energy-efficient solution improving over NSC and LSB by 23.3% and 17.9%, respectively. *CSB-TSO achieves the same energy efficiency as CSB-RC while offering stronger consistency.*

C. Performance Implications

Stalls due to the store buffer. The processor stalls when the store buffer is full and there is a store operation at the head of the reorder buffer (RoB), or when the store buffer is not empty and there is an atomic operation at the head of the RoB. Figure 16 shows the resulting stalls for the configurations studied. Compared to a unified SQ/SB, a partitioned SQ and SB may increase stalls (this effect in evident in some cases when comparing NSB to LSB). However, stalls are reduced on average as we allow: i) more coalescing, or ii) out-of-order writes in the case of CSB-RC. The former is evident in the difference between LSB and CSB-TSO (from 5.5% to 4.9%, on average). Occasionally, keeping more stores in the store buffer for coalescing increases the contention for store buffer entries. In such a case, coalescing does not notably reduce stalls, although still saves energy.

Execution time. Figure 17 shows normalized execution time. The main performance improvements come directly from the reduction in processor stalls. On the other hand, hitting more in the store buffer (1 cycle) yields performance benefits with respect to hitting in the cache (4-cycles, pipelined). On average, CSB-TSO outperforms NSB and LSB by 6.2% and 5.2%, respectively, while CSB-RC by an additional 2%.

D. Cache Coherence Implications

While an atomic group write is in progress, cachelines that are written are locked. Conflicting permission demands are blocked on locked cachelines while permission prefetch requests are `NACK`'ed. If requests are inordinately delayed the cache coherence protocol may increase the average miss latency. Fortunately, this is not the case because conflicts are extremely rare. The application with the most contention, by far, is x264 with only around 500 blocked/`NACK`'ed requests per million memory operations.

VI. Related Work

Our work is largely inspired by concepts introduced in the following works [10], [7], [5], [8], [25], [26], [6], [17]. We aim to show that store→store reordering in TSO, to the extent it is imposed by coalescing, can be achieved $non-speculatively$. Similarly to much prior work, our approach reacts to data races (conflicts) but does not roll-back, eliminating the complexity and the uncertainty involved in other approaches. In this section, we describe related work and point out the differences and how we advance the state-of-the-art.

SSB and ASO. An example of speculative coalescing is the Scalable Store Buffer (SSB) [5] that maintains all stores in a large FIFO (for replay if needed), called TSOB, and coalesces directly in the L1 which holds speculative state invisible to the coherence protocol. TSOB drains directly into the coherent L2 where it makes the stores, one-by-one, visible to the memory system. The TSOB can be large enough (on the order of 1024 entries) that its size compares to the L1. L1 evictions cause the store buffer to drain and external invalidations cause a partial replay of the stores to the L1. The primary benefit of the SSB is that loads can match (coalesced) stores directly in the L1, without resorting to an associative search of the TSOB. In the same work, Wenisch et al. propose Atomic Store Ordering (ASO) to speculatively relax all memory access order [5] relying on the SSB to provide the speculative state for the stores.

Our approach focuses on providing non-speculative coalescing in the store buffer by ordering the writes in a way that they appear atomic but do not deadlock. The major difference with SSB is that we reduce the number of writes to the L1, while SSB still performs a large number of writes in both the L1 and the L2. Our results, however, are not directly comparable, as SSB is shown to perform very well in write-intensive workloads that can benefit from very large store buffers and where store propagation may be less of an issue, while in our case, relatively small store buffers cover a significant portion of the coalescing potential of our workload (Figure 1).

TCC and BulkSC. Transactional Consistence and Coherence (TCC) [7] proposes a replacement of the typical directory-based coherence protocol with one where a group of writes are broadcasted atomically over the network and invalidate (or update) cachelines in private caches. Because of this, TCC requires significantly more network bandwidth than a typical system [7]. Atomic broadcasts squash transactions that have speculatively read a conflicting value. Our approach of atomically writing a group of cache lines is compatible with the typical directory-based invalidation protocols, results in far fewer writes, and naturally supports permission prefetching for performance. We believe that it can form the basis of an alternative implementation of TCC that would make it more practical.

BulkSC is a transactional (speculate-and-rollback on conflict) approach to Sequential Consistency (SC). Instruction "chunks" are executed speculatively and committed atomically in absence of any conflict on the speculative state [8]. Detection of (read and write) conflicts and enforcement of atomicity for group writes is entrusted to a centralized arbiter (or a hierarchy of arbiters for a distributed implementation) [8]. BulkSC offers an alternative implementation of the mutual-exclusion approach to atomic group writing. The difference with our approach is that we do not need to first obtain permission for writing (which can increase latency), but instead atomicity is based on the ordering of the writes.

Oklahoma. As far as we know, there is only one *distributed* proposal to atomically write a group cachelines without resorting to broadcast or to a centralized arbiter, but the solution is also *transactional* (widely considered as precursor of transactional memory [27]). It is a generalization of the load-linked/store-conditional, using reservations and a two-phase "*all-or-nothin*" commit of writes (using address-order for the writes), called the Oklahoma update [9]. Because it is an all-or-nothing solution, it suffers from livelock, which can only be solved if an additional aging mechanism is added on top. Rajawar and Goodman propose such a transactional approach using timestamps in [10]. Lastly, the Oklahoma update, does not address deadlocks arising from resource conflicts (e.g., limited cache or directory associativity resulting in deadlocking cache or directory evictions).

The challenge is to guarantee deadlock and livelock freedom. To the best of our knowledge there has been no prior solution to allow *non-speculative, incremental* update (one cacheline at a time) but still preserve atomicity, avoid deadlock, and avoid livelock by guaranteeing forward progress in the face of conflicts.

Racer. One of the closest work to ours is Racer [6]. Racer implements a coherence protocol with self-invalidation that is triggered when a read-after-write (RAW) race is detected in a race detector (called RAWR) co-located with the LLC. Racer also performs self-downgrade for the stores via a coalescing store buffer by sending the writes to a structure, called OSO, also co-located with the LLC. However, OSO on by itself does not provide any kind of deadlock avoidance. This means that groups with conflicting writes must be serialized in the OSO (groups with non-conflicting writes can interleave their writes in any desired way). While no detailed solution is provided in [6], prior solutions would fit the OSO case, if one is willing to accept the overhead: writing to the OSO with mutual exclusion à la TCC [7] or gaining permission first à la BulkSC [8]. Our solution would be ideal for Racer as there would be nothing more to do than simply write to the OSO in lex order. The *minimum common address conflict* provides the atomicity guarantee between conflicting atomic groups.

Non-speculative Load-Load Reordering. Recently, Ros et al. proposed a non-speculative mechanism to reorder loads in TSO [17]. Their approach is based on *blocking conflicting stores* in their store buffers until an ongoing load-load reordering on another core can no longer be observed. In contrast, we propose non-speculative store-store reordering by *blocking conflicting loads* and guaranteeing that reordered stores appear atomic with respect to other conflicting stores. Absence of deadlock is guaranteed in a mirror way in the two works: In non-speculative load-load reordering, loads are guaranteed not to block even though conflicting stores are blocked [17]. In our work, stores are guaranteed not to block even though conflicting loads are blocked. On first sight the two approaches may seem incompatible with respect to deadlock. The challenge of reconciling these two approaches into one unified scheme is left for future work.

VII. Conclusion

There are situations where out-of-core speculation may not be available or practical (e.g., high-efficiency designs); or situations where invasive changes to the memory hierarchy, the coherence protocol, etc., may not be acceptable. For such cases, providing store coalescing in TSO was an unsolved problem. In this work, we propose a non-speculative, deadlock-free, approach to TSO coalescing. We show that our TSO solution reaches performance and energy savings close to the case where we relax store order (i.e., RC). In a broader sense, in this paper, we give a new *non-transactional* solution to the problem of writing a group of cachelines as an atomic unit. This can positively impact a number of other proposals that require this functionality.

Acknowledgment

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

References

[1] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers (TC)*, vol. 28, no. 9, Sep. 1979.

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

[3] C. SPARC International, Inc., *The SPARC Architecture Manual (Version 9).* Prentice-Hall, Inc., 1994.

[4] 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, Jul. 2010.

[5] 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.

[6] A. Ros and S. Kaxiras, "Racer: Tso consistency via race detection," in *49th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2016.

[7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *31st Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2004.

[8] 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.

[9] 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, Nov. 1993.

[10] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," in *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2002.

[11] 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.

[12] T.-F. Tsuei and W. Yamamoto, "Queuing simulation model for multiprocessor systems," *IEEE Computer*, vol. 36, no. 2, Feb. 2003.

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

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

[15] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Computing Surveys*, vol. 3, no. 2, Jun. 1971.

[16] E. W. Dijkstra, "Hierarchical ordering of sequential processes," *EDW-310, E.W. Dijkstra Archive, Center for American History, University of Texas at Austin*.

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

[18] S.-W. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers (TC)*, vol. 49, no. 11, 2000.

[19] K. Skadron and D. W. Clark, "Design issues and tradeoffs for write buffers," in *3rd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 1997.

[20] N. P. Jouppi, "Cache write policies and performance," in *20st Int'l Symp. on Computer Architecture (ISCA)*, May 1993.

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

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

[23] 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.

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

[25] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram, "Efficient sequential consistency via conflict ordering," in *17th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2012.

[26] D. Gope and M. H. Lipasti, "Atomic SC for simple in-order processors," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014.

[27] J. R. Larus and R. Rajwar, *Transactional memory*. Morgan & Claypool Publishers, 2007.