

PhasedStore: Supporting High-performance Write-through Cache-coherence Protocols under TSO

Burak Ocalan
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA
bocalan2@illinois.edu

Chloe Alverti
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA
xalverti@illinois.edu

Shashwat Jaiswal
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA
sj74@illinois.edu

Antonis Psistakis
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA
psistaki@illinois.edu

David Koufaty
Unaffiliated
dkoufaty@gmail.com

Suyash Mahar
University of California
San Diego
San Diego, California, USA
smahar@ucsd.edu

Steven Swanson
University of California
San Diego
San Diego, California, USA
sjswanson@ucsd.edu

Josep Torrellas
University of Illinois at
Urbana-Champaign
Urbana, Illinois, USA
torrella@illinois.edu

Abstract—Current multiprocessors that support the total store order (TSO) memory consistency model invariably use write-back (WB) cache-coherence protocols. When their hardware needs to issue write-through (WT) stores as in uncached operations, their performance may suffer: writes to main memory have to be fully serialized, potentially forcing the program to observe the full latency of round trips to memory.

To solve this problem, this paper presents a novel architecture that supports high-performance cache-coherent WT stores under TSO. The architecture, called *PhasedStore*, extends the store queue in the cores and the directory. Individual WT stores fully overlap with other stores and still satisfy TSO.

PhasedStore is useful in environments that require a WT cache-coherence protocol. This can be the case in resilience-critical platforms where node failures should not cause the loss of shared program state, or platforms with CPUs and accelerators where programs follow a producer-consumer pattern. This paper evaluates PhasedStore in the first environment, namely a CXL-based distributed shared-memory platform where shared data in the program uses a WT protocol to enable recovery. Our evaluation shows that PhasedStore is very effective. Compared to using the conventional approach to implement WT under TSO, PhasedStore reduces the average execution time of a set of parallel applications by 1.88x.

I. INTRODUCTION

Multiprocessors with cores that support a TSO memory consistency model [36], [41], like those from Intel [24] and AMD [3], use write-back (WB) cache-coherence protocols. When these cores need to issue write-through (WT) stores, as in uncached operations, the requirement of TSO to enforce full serialization of the writes is particularly onerous. Indeed, each WT store involves a round trip to main memory (even if there are no sharers), which needs to complete before the next store can be issued. As a result, a core’s store queue quickly fills up and the program may end up seeing the full latency of round trips to main memory on the critical path.

When WT stores are rare, this overhead is tolerable. However, some future environments may use frequent WT stores and, therefore, would benefit from an efficient WT imple-

mentation under TSO. Two such environments are resilience-critical platforms and machines with CPUs and accelerators.

As an example of the first environment, consider the emerging Compute Express Link (CXL) framework [12], [15], [38], [27]. A CXL-based distributed shared-memory cluster can improve its resilience by supporting a WT cache-coherence protocol for critical application data. In such a platform, a software or hardware fault can cause the crash of a compute node (CN) participating in a shared-memory application [31], [47], [48]. With a WB coherence protocol, a failed CN will result in the permanent loss of cached dirty shared data, making program recovery next to impossible. With a WT coherence protocol for critical shared data, writes to such data propagate to the more resilient memory nodes (MNs). Hence, CN failures result in no loss of such data, which enables program recovery.

The second environment is systems integrating CPUs and accelerators. To attain high performance and programmability, these accelerators may be cache-coherent with the CPUs [46], [28], [32], [20]. In this case, supporting an efficient WT primitive will be useful for two reasons. The first reason is that many programs in these environments may follow a producer-consumer pattern, where data generated by one compute engine is then consumed by another one. In this case, supporting WT cache coherence will improve performance.

The second reason to support a WT primitive is to simplify the accelerator hardware. Designing accelerators that support an advanced cache coherence protocol like those of CPUs is expensive. However, if the CPU supports a WT primitive, the accelerator design is simplified: the accelerator only needs to read from memory rather than from another engine’s caches—and, of course, accept incoming invalidations and write back its cached dirty data. For these reasons, standards that target coherent environments with CPUs and accelerators such as CXL and Arm’s AMBA CHI [4] include WT protocols.

To address these challenges, this paper presents a novel architecture that provides, for the first time, high-performance

cache-coherent WT stores under TSO. The architecture, called *PhasedStore*, enhances the store queue (SQ) in cores and the directory to support a *two-phase* WT store. In the first phase, called *Seal*, the store marks the corresponding directory entry as unavailable to other cores and invalidates all other sharer nodes. In the second phase, called *Unseal*, the hardware updates the directory, memory, and local caches, and marks the directory entry as available to other cores. While a basic WT store now requires two round trips to the memory system for Seal and Unseal, WT stores can be fully overlapped with each other. Moreover, PhasedStore introduces a write-coalescing optimization to reduce the number of messages.

The paper presents the PhasedStore design, its interaction with WB stores, optimizations that combine multiple WT stores, and other design decisions. We evaluate PhasedStore in a scenario that models the first environment described above: a resilience-critical platform. Specifically, we simulate a CXL cluster with 16 4-core CNs and 16 MNs. The cluster runs parallel applications where shared data is allocated in the CXL tier in the MNs and uses a WT protocol, and the rest of the data is allocated in the memories of the CNs and uses a WB protocol. The evaluation shows that PhasedStore is very effective. Compared to using the conventional approach to implement WT under TSO, PhasedStore reduces the average execution time of the applications by 1.88x.

This paper makes the following contributions:

- PhasedStore, a new architecture that provides high-performance WT cache-coherence protocols under TSO.
- An evaluation of PhasedStore in a simulated CXL cluster with 16 4-core CNs and 16 MNs, and its comparison to other WT/WB cache-coherent environments with different consistency models.

II. BACKGROUND

A. Total Store Order and Write-Through Caching

The Store Queue (SQ). Figure 1 shows the operation of a common implementation of the store queue (SQ). The processor front-end decodes instructions in program order and, on finding a store, inserts it on both the reorder buffer (ROB) and the SQ. This is shown for ST E ①. When the store reaches the ROB head and both its address and data are computed, the store retires. This is shown for ST C ②. Retiring involves advancing the Retirement Pointer in the SQ and, potentially, initiating the merging of the store into the memory subsystem—if the memory consistency model allows it. In the figure, ST A and ST B are already retired and, therefore, may potentially be in the process of merging into the memory subsystem ③. Once a store is fully merged, it exits the SQ.

Total Store Order (TSO) Memory Consistency Model. The TSO model (e.g., SPARC TSO [41] and x86 TSO [36]) requires store→store order. Therefore, all the retired stores in Figure 1 (ST A and ST B) are merged with the memory subsystem in strict FIFO sequence, without overlapping with each other. This is not the case in more relaxed models, such as Weak Consistency (WC) or Release Consistency

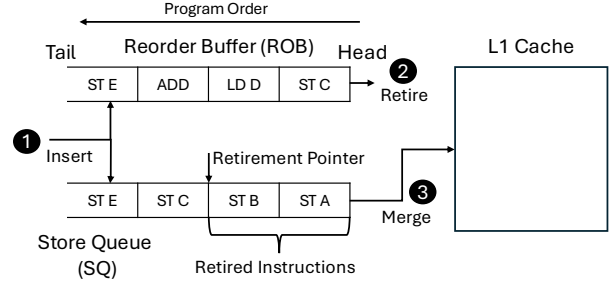


Fig. 1: Operation of a conventional store queue.

(RC) [19], which do not enforce store→store ordering to different addresses. In their case, all the retired stores to different addresses can be merged in parallel or out of order.

TSO does not require store→load order to different addresses. Hence, while a store waits to be merged, subsequent loads to a different address can retire when they reach the ROB head. This occurs to LD D in Figure 1 after ST C retires. Finally, TSO requires load→load order, which would imply that loads are issued to memory in program order. In reality, cores can issue loads to memory in parallel or out of order, speculatively [18].

TSO and WB stores. In WB cache-coherence protocols, before a store can be merged with the memory subsystem, the target line has to be brought into the L1 cache in exclusive state. Fetching the line into the L1 in this state when the store is allowed to merge can add substantial overhead on the critical path. To eliminate such overhead, as soon as possible after a store is inserted into the ROB, the core issues an exclusive prefetch in hardware for the store address [18]. Hopefully, by the time the store tries to merge into the memory subsystem, the line will be in exclusive state in the L1 and merging will simply involve updating the L1 cache.

TSO and WT stores. In WT cache-coherence protocols, merging a store with the memory system under TSO is expensive. The store must propagate all the way to the main memory and an ack must return to the core. This round trip cannot be overlapped with the merging of earlier or later stores. Exclusive prefetching is typically not useful—although in some cases it could hide the latency of invalidating sharer caches. As a result, a WT store under TSO blocks at the head of the SQ for an entire round trip to main memory, which can last hundreds of ns. If, because of the blocking, the SQ fills up and a new store cannot be inserted, the core stalls.

B. Compute Express Link (CXL)

CXL is a standard for interconnecting multiple compute nodes (CNs) and memory nodes (MNs) [12], [15], [38], [27] that allows attached memory to be configured as either node-private or shared without cache coherence, or shared with cache coherence. Our focus is on the latter, which enables a distributed shared-memory cluster (CXL-DSM) of multiple CNs and MNs, where the CNs share memory with cache coherence at the granularity of cache lines [12]. Figure 2 shows such a system with two multi-core CNs and two MNs.

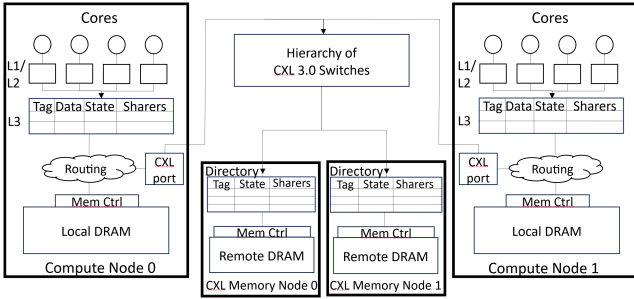


Fig. 2: Overview of a CXL-enabled distributed shared memory system (CXL-DSM).

Each MN has a portion of the remote (i.e., CXL) shared memory with a directory that specifies which CNs have copies of memory lines in their caches and in what state. We call the directory *remote directory*. Each CN has its own private local memory and a cache hierarchy that can contain lines from the local memory and from the remote shared memory. Requests to remote memory are routed through a CXL port. In each CN, the directory in the L3 specifies which local cores cache copies of remote or local memory lines and in what state. The coherence protocols for local data and for remote shared data can be different.

III. NEED FOR EFFICIENT SUPPORT FOR WT

In this paper, we show how to efficiently support cache-coherent WT stores under TSO. This is a primitive that can be useful in important future scenarios. In this section, we describe three such scenarios: resilience-critical platforms, machines with CPUs and accelerators, and systems equipped with persistent memory. We consider each in turn.

A. Resilience-Critical Platforms

A CXL-based distributed shared-memory cluster [12], [15], [38], [27] can improve its resilience by supporting a WT cache-coherence protocol for critical application data. In such a cluster, faulty software or hardware can crash a CN participating in a shared-memory application [31], [47], [48]. Note that MNs are more robust [45], as failures can be tolerated with replication [44], by persisting to non-volatile memory, or by using more expensive hardware.

With a WB coherence protocol, a failed CN will result in the permanent loss of cached dirty remote data, which will make it very hard to recover the program. Persistent caches (e.g., Intel’s eADR[25] or CXL’s Global Persistent Flush [12]) enable the automatic flushing of dirty data, but are effective only for power failures [48].

With a WT coherence protocol, writes propagate to the resilient MNs. Hence, CN failures result in no loss of program shared state, which enables program recovery. Unfortunately, supporting high-performance WT under the common TSO memory consistency model is hard. Indeed, to ensure that multiple writes from a core are observed by other cores in program order, the straightforward design requires full serialization of the writes’ round trips to main memory. Such costly

serialization is not required if the system supports a weaker form of consistency such as Release Consistency (RC) [19], relies on hardware-supported transactions [31], or implements software transactions [21], [47]. Our goal, however, is to support high-performance WT execution under the more popular TSO model, and for general-purpose, non-transactional, shared-memory programming environments. How to do it is an open problem.

We believe that, for such a WT environment to deliver high performance, the shared data that is allocated in CXL and is needed to recover the application on a failure should receive only a modest fraction of the total processor requests. The majority of the requests should be directed to private data that is allocated in the local memory of the nodes and that is not critical for recovery. Such data can use a WB protocol within each node. These assumptions are consistent with the literature [47], [21].

B. Machines with CPUs and Accelerators

The current trend of computing platforms that integrate CPUs and accelerators [26] is likely to grow. To attain high performance, accelerators may have caches and, to retain programmability, accelerators may be cache-coherent with CPUs [46], [28], [32], [20]. In this scenario, supporting an efficient WT primitive will be useful for two reasons.

First, in many of these environments, programs may follow a producer-consumer pattern [46], where data generated by one compute engine is then consumed by another one. For example, consider an image processing application. After loading an image, the application may invoke CPUs and different accelerators in sequence to perform operations on the image. In this case, supporting an efficient cache-coherent WT primitive may improve performance.

The second benefit of a WT primitive is to simplify the accelerator hardware. Designing an accelerator that supports an advanced cache-coherence protocol like those of CPUs may be expensive. However, if the CPU supports a WT primitive, the accelerator design is simplified. The accelerator only needs to read from memory rather than from the cache of another engine. Of course, the accelerator must still accept incoming invalidations and write back its cached dirty data.

For these reasons, standards that target coherent environments with CPUs and accelerators such as CXL and Arm’s AMBA CHI [4] include WT protocols.

C. Systems Equipped with Persistent Memory

In systems with byte-addressable persistent memory (PM), configuring selected cache levels as WT can provide faster durability, since every store is immediately pushed to the persistence domain. Prior work on PM frequently assumes WB caches, enforcing durability through cache-line flush instructions (e.g., `clflush`) and fences, or by using non-temporal stores that bypass the cache [2]. However studies [5] show that WT caching can avoid per-line flushes and fence overheads—making WT a viable and potentially higher-performance mechanism for durable environments. To this

end, high-performance, TSO-consistent, WT caches can be beneficial to applications with persistent heaps [11].

D. Environment Used in our Design and Evaluation

Our PhasedStore design and evaluation in this paper assumes the environment described in Section III-A. Specifically, we consider a CXL cluster running parallel applications where shared data is allocated in the CXL tier and uses a WT protocol, while the rest of the data is allocated in the memories of the nodes and uses a WB protocol. Implicitly, we assume that faulty software or hardware can crash a CN [31], [47], [48], but that MNs do not fail, similar to [45], [31], [47]. MNs with CXL memory are likely to be more robust than inexpensive CN nodes. In addition, MNs and CXL switches are expected to have their own independent power domain [45], and not run general-purpose software, which also causes CN failures.

IV. PHASEDSTORE: SUPPORTING WT IN TWO PHASES

In this section, we describe our design of PhasedStore. To put it in perspective, Section VI describes an alternative design that we discard because it is inferior.

A. Overview

Figure 3 shows the idea behind PhasedStore. Its goal is to overlap the execution of multiple WT stores, so that their latency is not serialized. To do so, it introduces two phases in the execution of a WT store: the *Seal* and *Unseal* phases.

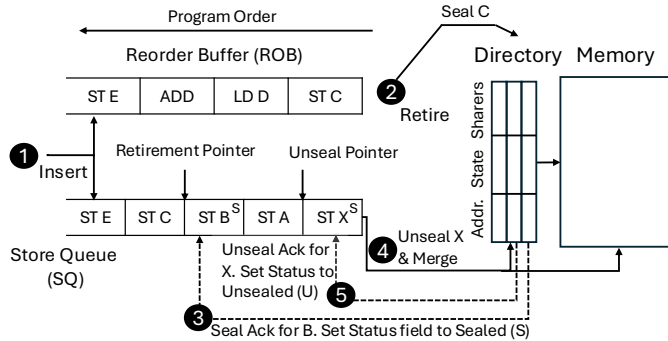
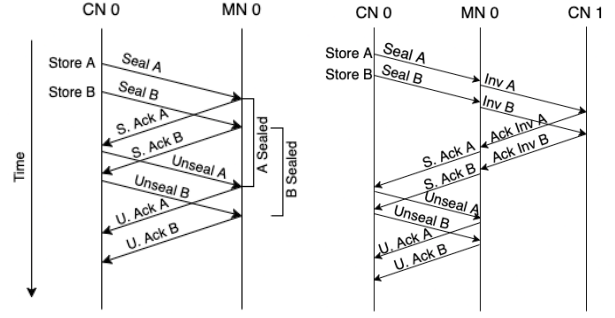


Fig. 3: PhasedStore overview.

- *Seal*. When a store retires from the ROB (e.g., ② for ST C), the hardware sends a Seal request to the corresponding directory module, which resides in a memory node (MN). As the request traverses the cache hierarchy, if a cache is found to contain the line, the line is made unavailable to future reads and writes. When the directory receives the Seal request, it seals the corresponding directory entry and sends invalidations to all the CNs of the machine with copies of the line. A sealed directory entry does not accept new requests for this cache line—it either buffers or nacks them. Once the directory receives the acks for all the invalidations, it sends a Seal Ack to the requester (e.g., ③ for ST B), and the hardware changes the *Status* field in the store’s SQ entry from *Idle* to *Sealed* (S).
- *Unseal*. When the SQ entries for a store and for all its older stores in the SQ are marked as *Sealed*, the hardware issues an Unseal request for the store (which includes the store’s data)



(a) *A* and *B* are not shared (b) *A* and *B* are in *CN 1*
Fig. 4: Timeline of two stores in PhasedStore.

to the directory (e.g., ④ for ST X). In addition, PhasedStore introduces a new pointer in the SQ called the *Unseal pointer*, which points to the oldest store that has not yet issued an Unseal request (e.g., ST A in Figure 3). When such store sends its Unseal request, the hardware advances the Unseal pointer’s position. We see, therefore, that Unseals are sent in program order.

When the directory receives the Unseal, it merges the store’s data with the MN’s memory and unseals the directory entry, allowing other cores to access the entry. It also sends an Unseal Ack to the requester with the updated line. The Unseal Ack causes the line to update each cache of the cache hierarchy of the requesting node, making the line accessible to other requests. When the Unseal Ack finally reaches the core (e.g., ⑤ for ST X), it will change the Status field of the store’s entry in the SQ from S to *Unsealed* (U). When an Unsealed store reaches the head of the SQ, it is removed from the SQ. We will see that sending the Unseal request for a store only after the store and all its older stores have received a Seal Ack guarantees TSO.

Figure 4 shows the timeline of two stores in PhasedStore. Compute Node CN 0 writes to *A* and *B*, where *A* and *B* are in different cache lines. Figure 4a shows the case when *A* and *B* are not cached in any other node. The figure shows the time when the corresponding directory entries are sealed. While each store requires two round trips, the two writes proceed in parallel. Figure 4b shows the case when *A* and *B* are cached in a second node (CN 1). In this case, the directory sends invalidations to CN 1.

B. Message Reordering Does not Violate TSO

The cache hierarchy of CNs and the CXL network can reorder the messages of multiple WT stores issued by the same node. In all cases, however, TSO is guaranteed. To see why, consider Figure 5, which shows the example of Figure 4a when Seal/Unseal are reordered (Figure 5a) and when Seal/Unseal Acks are reordered (Figure 5b).

Recall that Seal messages are issued in program order; the same is true for Unseal messages. As shown in Figure 5a, the reordering of Seal messages simply means that *B*’s entry in the directory is sealed before *A*’s entry. For a time interval, the old value of *A* can still be seen while *B* is already inaccessible.

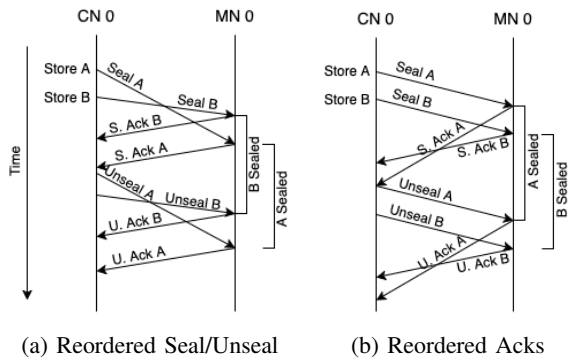


Fig. 5: PhasedStore guarantees TSO despite reordering.

This is acceptable in TSO, since the new value of B is not visible. Similarly, the reordering of Unseal messages means that, for a time interval, the new value of B is visible while A is inaccessible. Since no core can see both the new value of B and the old value of A , TSO is satisfied. This is guaranteed by the fact that the Unseal message for a store is issued only after the store and all its older stores have received Seal Acks.

Figure 5b shows the case when Seal Acks are reordered. This is harmless. Despite receiving Seal Ack for B first, PhasedStore waits for Seal Ack for A before issuing Unseal A and then Unseal B . Similarly, the network reordering of Unseal Acks does not violate TSO.

C. Hardware Requirements of PhasedStore

PhasedStore introduces new types of messages, and extends the SQ, caches, directory, and the CXL port.

Messages. While a conventional WT store needs a request-reply message pair, PhasedStore needs two message pairs: *Seal* plus *Seal Ack* (*S.Ack*), and *Unseal* plus *Unseal Ack* (*U.Ack*). *Seal* and *Seal Ack* carry the address (byte, word, etc) of the location written. *Unseal* carries the address of the location plus the data written. *Unseal Ack* carries the address of the line written plus the line data.

Store Queue. Figure 3 shows how PhasedStore extends the SQ. It introduces the *Unseal pointer* to point to the oldest store in the SQ that has not yet issued an Unseal request. Let's call this store S_{no-u} and the oldest store not yet retired S_{no-r} . They are ST A and ST C, respectively, in Figure 3. All the stores older than S_{no-u} have issued Seal requests, received Seal Acks, and issued Unseal requests. All the stores older than S_{no-r} but no older than S_{no-u} have issued Seal requests and maybe received Seal Acks.

PhasedStore adds the 2-bit *Status* field to each SQ entry. It can be: Idle (I) if the store has not yet received the Seal Ack; Sealed (S) if it has received the Seal Ack but not the Unseal Ack; and Unsealed (U) if it has received the Unseal Ack. Stores older than S_{no-u} may have Status set to S or U, while stores older than S_{no-r} but no older than S_{no-u} may have Status set to I or S. Stores no older than S_{no-r} have Status set to I.

As will be shown in Section IV-E, PhasedStore also adds a *Last* bit to each SQ entry to enable store coalescing.

Caches. A Seal request traverses the cache hierarchy. For each cache, if the cache has the corresponding line, Seal sets it to a transient state that prevents future reads or writes. If the cache does not have the line, Seal allocates a new entry in the cache. The Seal request eventually reaches a MN. The Unseal Ack brings the updated cache line data from the MN to the core, traversing the caches. In each cache, Unseal Ack writes the line, marks it as Shared, and clears the transient state.

Directory. In conventional directories, when a write request reaches the directory, the directory entry transitions to a transient state that prevents new requests from accessing that entry until all sharer caches have been invalidated, the acks have been received in the directory, and a response has been sent to the requester. With PhasedStore, this period when the directory entry is inaccessible extends until the directory sends the *Unseal Ack* to the requester. One way to support this functionality is with a *Sealed* bit per directory entry that is set while the directory entry is sealed. If a Seal request misses in the directory, a new directory entry is allocated. If all the entries in the directory set are Sealed, the Seal request is either temporarily buffered or returned to the sender with a nack.

Per-core queues on the CXL port. As will be shown in Section IV-F, PhasedStore also extends the CXL port with queues per core to minimize the reordering of Seal messages.

D. PhasedStore and Write-Back Stores

PhasedStore can operate in an environment where some data is accessed with a WB cache-coherence protocol. This may be the case, for example, for data allocated in the local memories of the CNs in Figure 2. In the following, we refer to the data accessed through PhasedStore as *remote* data and the data accessed through a WB protocol as *local* data. The SQ of a core can contain a mix of both local and remote stores, and TSO is supported for both types of stores.

To understand the operation in this environment, consider a local store L in the SQ surrounded by older remote stores R_o and younger remote stores R_y . To ensure TSO, the following two conditions must hold. First, before L can be merged with memory, the old values that R_o overwrite must be made inaccessible to other cores. Second, the new values that R_y generate can be accessible to other cores only after L is merged with memory. To support the first condition, before L is merged with memory, R_o must have received their Seal Acks—which guarantee that R_o 's directory entries have been sealed and, therefore, that the old values that R_o overwrite are inaccessible. To support the second condition, L must be merged with memory before R_y issue Unseals—since Unseals will make R_y 's updates visible to other cores.

Figure 6 shows how the SQ supports this algorithm. Assume that memory locations A, C, and D are remote, while locations B and E are local. First, like in conventional systems, after L is added to the SQ (e.g., ① for ST E), as soon as its target address is known, the hardware issues an exclusive prefetch ②. Then, when L is pointed to by the *Unseal pointer* (e.g., ③ for ST B), L can be merged with the memory subsystem ④. This is because all the older remote stores have received

Seal Acks, ensuring that the old values are inaccessible (and, in fact, sent Unseals). Later, after L receives the ack for merge completion ⑤, the Unseal pointer is shifted left one position ⑥. This allows the subsequent remote store (i.e., ST C) to send its Unseal request (if it has already received its Seal Ack), to make its new update visible.

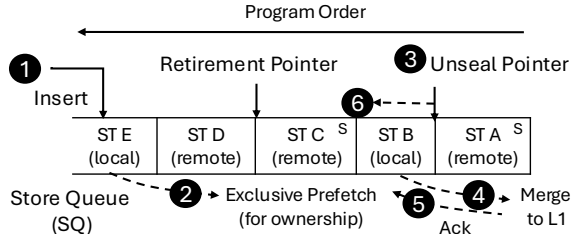


Fig. 6: Supporting local stores in PhasedStore.

Consecutive local stores are merged to the memory system in strict sequential order—i.e., the second store can only be merged after the first one is merged. This is required for TSO.

For this support to work, we need to determine whether a store is remote, and therefore uses a WT protocol, already in the SQ. One way to accomplish this is to define a new WT memory type associated with page tables. Applications would use the new memory type for remote data accesses.

E. Coalescing WT Stores to the Same Line

A common access pattern involves a core issuing multiple consecutive stores to the same cache line. Such stores can target the same address (byte, word, etc) or different addresses in the same line as in streaming accesses. Existing work using WB protocols coalesces these stores into a single one to reduce traffic and improve performance [35]. In the WT environment of PhasedStore, we would also like to coalesce such consecutive WT stores.

Without any extra support for coalescing, PhasedStore executes this pattern inefficiently. To see why, consider a core writing to address A and then $A+4$ in the same cache line. As shown in Figure 7a, the core issues Seal A and, immediately after, Seal $A+4$. Upon receiving Seal A , the L1 cache disables further reads/writes to the cache line. This causes Seal $A+4$ to temporarily block. Seal A is followed by S.Ack A , Unseal A , and U.Ack A . When the latter reaches L1 and updates the line, Seal $A+4$ is unblocked and proceeds to perform its own two round trips. Thus, the hardware serializes the stores.

To prevent this serialization, PhasedStore coalesces consecutive WT stores to the same cache line in the SQ. The result is shown in Figure 7b. The core only issues Seal A . Once S.Ack A is received, the core traverses the SQ, collecting all subsequent stores to the line, and then sends the combination of all the line updates in the Unseal A message. Unseal A updates memory and U.Ack A updates all the cache levels.

To support this mechanism, PhasedStore adds a *Last* bit to each SQ entry. If set, it marks the last store in a sequence of consecutive stores being coalesced. Figure 8a shows an example with four consecutive stores to different addresses ($A, \dots, A+12$) of line L_A . When the Retirement pointer (RP)

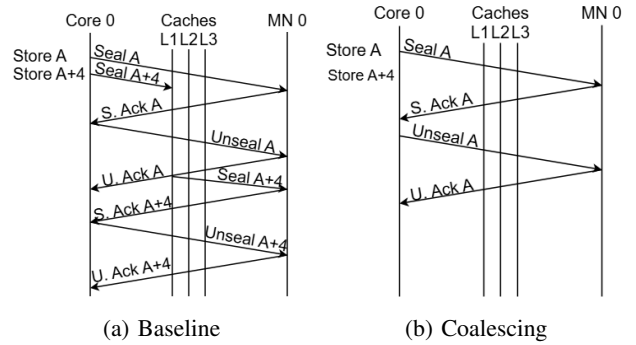


Fig. 7: Consecutive stores to same cache line.

reaches the first store in the batch (ST A), the hardware retires the store, issues Seal A , and advances the RP. Then, ST $A+4$ retires and RP advances again but, since there is already an outstanding Seal for line L_A , no new Seal is issued. The same occurs for ST $A+8$. At the same time, PhasedStore accumulates all the updates to the line in order.

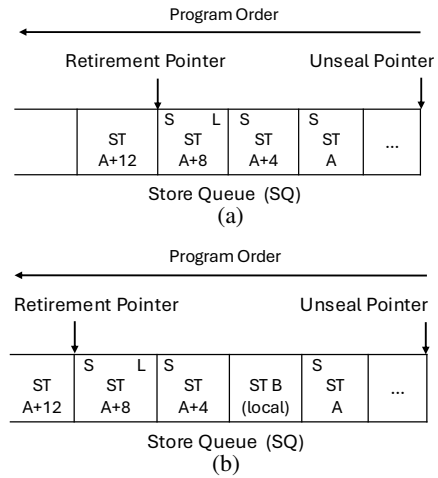


Fig. 8: Coalescing stores to the same cache line.

Assume that, at this point, the response S.Ack A arrives. The hardware scans the SQ, coalescing all the consecutive stores to line L_A that have already retired (ST A , ST $A+4$, ST $A+8$) by: (1) setting the Status bit of all such stores to S and (2) setting the Last bit of the last one of such stores (ST $A+8$) to 1 (shown in Figure 8a with an L). Later, when the Unseal pointer (UP) reaches ST A , the UP will advance without issuing Unseal requests until past the entry with $L=1$ (ST $A+8$). At that point, the hardware will issue a single Unseal A message with the combined three updates. In the figure, when the next store (ST $A+12$) retires, the hardware will again issue a Seal message and potentially initiate a new coalescing action.

Coalescing WT Stores with Intervening Accesses to Other Lines. PhasedStore can coalesce a set of WT stores to the same line even if the set is interleaved with loads to different lines or with WB (i.e., local) stores to different lines. Consider first intervening local stores. Figure 8b shows an example where, in between a set of stores to line L_A , there is a store ST B to local line L_B . Recall that, for line L_A , the Seal message is sent when the *first* store in the Coalesce set (i.e., ST A) retires,

and the Unseal message is sent when the Unseal Pointer (UP) goes past the *last* store in the Coalesce set (i.e., ST A+8, the last retired store when S.Ack A was received). Moreover, ST B can only be merged with memory after ST A has received S.Ack A (Section IV-D). Hence, in the example, the order of messages is: (i) the core sends Seal A; (ii) the core receives S.Ack A, setting L for ST A+8; (iii) the core merges B into the memory subsystem. The core will later issue the Unseal A message.

With this order of messages, it is impossible that another core could observe the old version of L_A (i.e., before ST A) and the new version of L_B . This would be a violation of TSO. Instead, when another core can see the new version of L_B , it either cannot access L_A or sees the new version of L_A after the three updates. Moreover, it is not possible that another core could observe the old version of L_B (i.e., before ST B) and the new version of L_A after the three updates. Instead, if another core can still see the old version of L_B , it either cannot access L_A or sees the old version of L_A .

Hence, PhasedStore supports WT store coalescing with intervening local stores without breaking TSO, as it seals the target directory entry when the first store in the Coalesce set retires and unseals it when the UP goes past the last store in the Coalesce set. In contrast, TSO-compliant store coalescing with intervening local stores cannot be easily accomplished with WB stores. The reason is that coalescing a set of WB stores to a line could make all such stores globally visible at the time when the first store is ready to merge with the memory subsystem—before the intervening store is made visible. To support TSO, we may need to build upon complex speculation and rollback mechanisms [42] or sophisticated ordering enforcement [35]. For the same reason, TSO-compliant store coalescing with intervening local stores cannot be supported in conventional WT protocols.

In PhasedStore, we do not support WT store coalescing when the intervening stores to different lines are WT stores. The reason is that our resilience model of Section V requires that Unseal messages be issued in program order. From the description of Figure 8b, this would not be the case if the intervening ST B was a remote WT store. The reason is that there is a single Unseal A for the three updates to L_A .

PhasedStore also supports WT store coalescing with intervening *loads* to different lines. Using the implementation described above for coalescing WT stores, it can be concluded that the intervening loads will appear to execute before any of the stores in the Coalesced set. This is acceptable under TSO.

F. Deadlocks in PhasedStore

Seal Message Reordering Can Lead to Deadlocks. Two Seal requests from a core can be reordered in the cache hierarchy or in the CXL network. This can potentially lead to deadlocks. Figure 9a shows an example. A core in compute node CN 0 writes to A and B. While their Seal requests are issued in program order, assume that Seal B reaches the directory in memory node MN 0 first. The hardware seals the corresponding directory entry and returns a S.Ack B to

CN 0. At the same time, a core in CN 1 writes to B and A, and their Seal requests arrive in order to MN 0. Seal B is nacked by the directory, since the corresponding entry is sealed. Further, assume that the hardware successfully seals A’s directory entry and, therefore, replies with a S.Ack A to CN 1. Later, the directory receives the delayed Seal A from CN 0 and nacks it because the corresponding entry is sealed. At this point, the cores deadlock, as CN 0 repeatedly keeps trying to seal A and CN 1 keeps trying to seal B. Since, in each core, the hardware cannot issue an Unseal for the second store until the first store has at least successfully completed a Seal operation, no core can make forward progress.

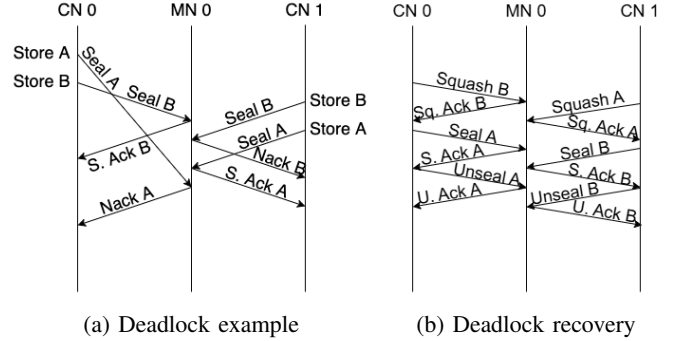


Fig. 9: PhasedStore deadlock and recovery.

Recovering from a Deadlock. While deadlocks may be infrequent, PhasedStore must detect and recover from them. Specifically, if a Seal message for the *oldest* WT store gets repeatedly nacked, after *DeadCount* retries, the issuing core hardware declares a deadlock. To recover from the deadlock, the core hardware stops retrying and, instead, sends squash messages for its younger stores that have successfully completed Seal operations. A squash message for a store unseals the corresponding directory entry. Then, PhasedStore reissues the first store without overlapping with younger stores. After the first store completes, the deadlock is broken and PhasedStore proceeds with the subsequent stores as usual.

Figure 9b shows the timeline of deadlock recovery. First, each core squashes its second store, which had sealed the directory. Then, the cores execute their first store without overlapping. After this, each core executes its stores as usual.

Minimizing Reorderings. In our analysis, we find that the reordering of a core’s Seal requests occurs more often due to interaction with another core in the same node than due to CXL network effects. Figure 10a gives an example of the pattern. Core 0 issues a Seal A ①, while Core 1 issues Seal A+4 ② (which references the same cache line) and then Seal B ③. We assume a sliced last-level cache (LLC). Seal A reaches the corresponding LLC slice first ④. It marks line L_A as inaccessible and proceeds to the CXL port of the node ⑤. Next, Seal A+4 from Core 1 reaches the LLC slice and blocks because the line is inaccessible ⑥. Seal B from Core 1 reaches the corresponding LLC slice ⑦, marks line L_B as inaccessible, and proceeds to the CXL port ⑧. Seal A+4 will

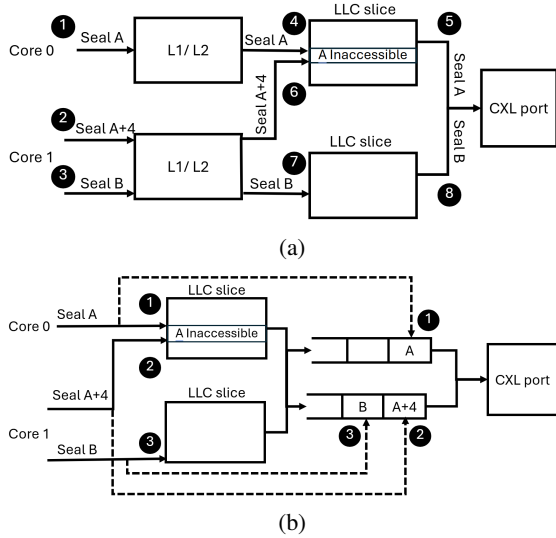


Fig. 10: Reordering of a core’s Seal requests due to interaction with another core in the same node (a), and hardware extension to prevent it (b).

only be unblocked when U.Ack A is received. We see that Core 1’s Seal A+4 and Seal B messages are reordered.

Such reorderings may increase the probability of deadlocks. Hence, PhasedStore includes a hardware extension that, at a performance cost, prevents these reorderings. The idea is shown in Figure 10b. The CXL port in the node has a queue per core. When a write from a core reaches an LLC slice, in addition to performing the usual operations (marking the line as inaccessible or blocking), it also claims an entry in the core’s CXL port queue. Requests in a queue are issued to the CXL network in order. Since a core claims the entries in its CXL port queue in order of request arrival to the LLC, the reordering described above will not occur.

Indeed, in Figure 10b, Seal A from Core 0 accesses its LLC slice and claims a position in Core 0’s CXL port queue (1). Then, Seal A+4 from Core 1 accesses the same LLC slice and blocks, but it claims a position in Core 1’s CXL port queue (2). Finally, Seal B from Core 1 accesses its LLC slice and claims the second position in Core 1’s CXL port queue (3). Seal B will not be issued to the CXL network until after U.Ack A is received and Seal A+4 is issued to the CXL network.

V. HANDLING COMPUTE NODE FAILURES

PhasedStore is designed so that, on a CN failure, the CXL-attached memory and the surviving nodes are left in a consistent state. Specifically, the CXL-attached memory is left in a state that corresponds to a valid program state—i.e., if the memory has been updated by a given write from a thread, it is guaranteed to also have been updated by *all the earlier writes* in program order from the same thread. Moreover, the state in the caches of the surviving nodes is consistent with the CXL-attached memory—i.e., caches are up-to-date and, furthermore, have no newer updates over those in memory. Such properties will make it easier for a software handler to recover the program after a failure.

The failure of a CN is detected by the classical approach of a Configuration Manager (CM) process that regularly sends heartbeat messages to all the nodes in the cluster [22], [1]. If a node does not respond within a timeout period, a node failure is flagged and recovery software is invoked. Zookeeper [22], [39], [40] uses a distributed algorithm that ensures the fault tolerance of the CM process.

We consider the main cases where a node CN_f can fail:

CN_f fails while receiving an invalidation message from CXL-attached memory. In this case, the directory module that sent the invalidation will never receive the ack and will stall. Then, the core that issued the write that triggered the invalidation will keep retrying and eventually assume that the write is in a deadlock. In this case, the core will perform the steps to recover from a deadlock (Section IV-F) and re-issue the write without overlapping. But such re-issued write will also not make progress and keep retrying. Retries will continue until the CM detects that CN_f failed and initiates a software handler to recover. Recovery is made easier by the fact that CN_f does not contain updates that are younger than those in the CXL-attached memory. The recovery will isolate CN_f and remove it from the sharer list of all directory entries. The CXL reset message [12] can be extended to support such functionality in hardware.

CN_f fails while having outstanding Seal requests. In this case, the directory will not receive the subsequent Unseal messages. Since some directory entries are sealed, subsequent Seal requests to such entries arriving from other nodes will not be serviced. Hence, such nodes will eventually keep retrying. Retries will continue until the CM detects that CN_f failed and initiates recovery. In the recovery, these outstanding stores will be aborted. As before, the CXL memory is left in a state that corresponds to a valid program state, and no caches have newer updates than the CXL memory. The recovery handler will isolate CN_f , remove it from sharer lists in the directories, and unseal the directory entries that were sealed by the incomplete updates. Neither directory nor CXL memory will be updated by the incomplete updates.

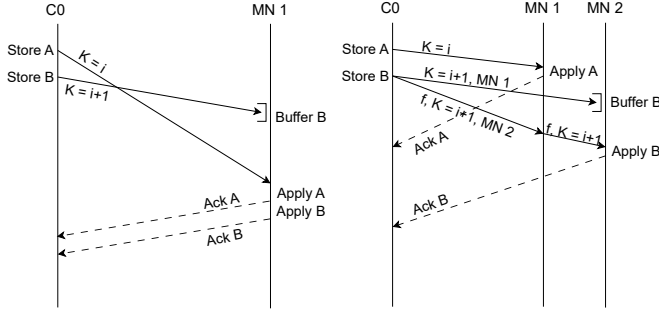
CN_f fails while having outstanding Unseal requests. These Unseal requests will reach the directories in the MNs. In each relevant MN, the controller will update the directory and memory, send a U.Ack to CN_f , and unseal the directory. The fact that CN_f will not observe the U.Acks is immaterial. Note that, because Unseal requests from a core are issued in program order (Section IV-A) and leave the node in program order (Section IV-F), irrespective of how many Unseal requests leave CN_f before CN_f fails, it is the older Unseals the ones that make it out. Hence, the CXL memory is left in a valid program state. The recovery handler will isolate CN_f and remove it from sharer lists in the directories.

VI. ALTERNATE DESIGN: ORDERING AT THE DIRECTORY

An alternate design to support WT writes under TSO can be deduced from [46]. The idea is for cores to pass program order information to the directory in each of the WT stores, so that the directory can apply the writes *in order*

and satisfy TSO. Specifically, each WT message from a core to the directory/memory includes a count K . Such count is incremented by the core hardware at every WT store.

Figure 11a shows an example where Core $C0$ performs WT stores to variables A and B , and both variables map to the same directory/memory node $MN1$. The ST A and ST B messages include the count $K=i$ and $K=i+1$, respectively. If the directory receives ST B first, it buffers it, and only applies it after applying ST A .



(a) WT stores to the same MN (b) WT stores to different MNs

Fig. 11: Passing ordering information to the directory.

Figure 11b shows the case when variables A and B map to different directory/memory nodes: $MN1$ and $MN2$, respectively. In this case, the hardware in $C0$ takes two additional actions when performing the WT to the second variable (B). First, it augments the ST B message with the ID of the MN that received the previous WT (the one for A). This will cue the directory in $MN2$ to wait for a message from $MN1$ before applying the update to B . Second, after sending ST B , the hardware in $C0$ sends an extra message to the MN of A . It is a forwarding message ($f=1$) that $MN1$ must forward to $MN2$ so that the latter applies the update with count $K=i+1$. When the directory in $MN1$ has both applied the update to A and received the forwarding message, it forwards the message to $MN2$, passing $K=i+1$. $MN2$'s directory will wait to receive the forwarding message before applying the update to B and then sending $Ack B$ to $C0$.

This design incurs several overheads. First, each core needs hardware to record, for each WT store, to which MN the store message travels. Such information is needed to detect consecutive WT stores to different MNs. Second, directories need a hardware structure to buffer and reorder WT stores that arrive out-of-order. Third, the two forwarding messages shown in the example are not acknowledged; hence, it is unclear what happens when a forwarding message is lost. Fourth, forwarding requires MN-to-MN messages, which are not supported in the current CXL protocol. Finally, when a counter overflows, the execution stalls until all in-progress WT stores complete and the counter is reset.

Unlike PhasedStore, this design cannot cause deadlocks. However, in our evaluation, we observe that deadlocks in PhasedStore are extremely rare. It can be shown that deadlock detection and recovery in PhasedStore take less than 0.1% of the overall execution time.

For these reasons, we regard this design to be inferior to PhasedStore and do not consider it further.

VII. CORRECTNESS

We use the Mur Φ [16] model checker to check the correctness of PhasedStore. Similar to prior work [46], we prevent state space explosion by limiting the validation parameters. We limit the number of CNs to three and the number of memory requests per CN to three. We model a single MN for the WT stores. Each CN has a local memory that is accessed with WB stores, while the WT stores access global memory. Our model checker includes speculative loads, store-to-load forwarding, and our PhasedStore extensions—i.e., the two-phased WT stores and aggressive store coalescing.

We test PhasedStore with 228 customized litmus tests designed to cover various scenarios:

- Case where all nine requests are WT stores. In these tests, we mainly verify the correctness of our two-phase protocol described in Section IV-A. Our tests include memory accesses to both the same and different cache lines.
- Case where each CN issues a WT store, a WB store, and a WT store. These tests put one WB store between two WT stores in order to verify the correct ordering of WB stores with respect to both prior and later WT stores. Additionally, we verify the aggressive store coalescing of two WT stores to the same cache line when there is a WB store in between.
- Case where we interleave a load, a WB store, and a WT store in all CNs with various orders, investigating the loads' integration with the store operations. Our verification includes speculative loads and store-to-load forwarding.

All of the tests pass, providing a bounded proof that PhasedStore supports TSO.

VIII. METHODOLOGY

Architecture. We model a CXL-enabled distributed shared memory architecture (CXL-DSM) like the one in Figure 2. Table I shows the configuration of the architecture. There is a single CXL switch connecting all CNs to all MNs. All round trip (RT) latencies are measured from a core. All caches have the same cache line size.

TABLE I: Modeled CXL-DSM cluster.

System	16 CNs, 16 MNs, 4 cores/CN at 2.4 GHz
Core	128-entry load queue; 72-entry store queue
CN L1 Cache	48KB, 12-way, 5 cycles RT, 64B line size
CN L2 Cache	512KB, 8-way, 13 cycles RT
CN L3 Cache Slice	2MB, 16-way, 36 cycles RT min.
CN Local Memory	45 ns memory access beyond L3, 410 GB/s bwdth, 128 GB size
CXL Memory	245 ns RT (200ns RT in CXL network + 45ns memory access), 160 GB/s bandwidth [30]

Table II lists the coherence protocols evaluated. Our proposed protocol (*PhasedStore-TSO*) is compared to conventional WB and WT protocols supporting TSO (*WB-TSO* and *WT-TSO*), and conventional WB and WT protocols supporting weak consistency [19] (*WB-WC* and *WT-WC*). In all protocols,

TABLE II: Cache coherence protocols evaluated.

PhasedStore-TSO	Proposed WT protocol. Has states S, I. Uses WB MESI protocol for local accesses. Supports TSO.
WB-TSO	Conventional WB MESI protocol. Supports TSO.
WT-TSO	Conventional WT protocol. Has states S, I. Uses WB MESI for local accesses. Supports TSO.
WB-WC	Conventional WB MESI protocol. Supports weak consistency (WC), where all non-synchronization accesses can reorder.
WT-WC	Conventional WT protocol. Has states S, I. Uses WB MESI for local accesses. Supports WC.

the data allocated in node memories (i.e., local data) uses a WB MESI protocol.

All WB protocols use exclusive prefetching for writes. In addition, all non-PhasedStore protocols use non-speculative store coalescing [35], [34], which coalesces local or remote stores to the same line when they are not interleaved by a store to another line. In PhasedStore, local stores are coalesced following the same pattern. However, remote stores use the more aggressive coalescing of Section IV-E, where they coalesce even if interleaved by local stores to different addresses.

In PhasedStore, the sizes of the protocol messages are as follows. Seal, Seal.Ack, Inval, Inval.Ack, and Read_request messages carry the line address and the CN ID. The Unseal messages carry the same information as Seal plus the updated data and a mask that indicates what bytes in the line are updated. The Unseal.Ack and Read_response messages carry the same information as Seal plus the up-to-date cache line.

In our evaluation, PhasedStore uses Nacks plus retries without backoff. DeadCount is set to 1,000. Nacks do not cause livelock: the owner process always makes progress. Nacks could theoretically cause starvation if the hardware is designed with biases. Our simulator does not introduce biases. The alternative is to use buffering at the directory, but this is arguably a more expensive solution: the buffers must be sized for the worst conditions. We also simulated the buffering approach. We used 128-entry buffers and never saw overflows. The performance with either approach is practically the same.

Evaluation Methodology. We evaluate our design using the SST simulator [33]. We generate the traces using Pin [23]. The traces do not have timing information, but include all instruction and data accesses, and synchronizations (lock acquire request, lock release, barrier arrive, and barrier leave). In the simulations, each core advances time by executing its own thread’s instructions. We ensure that the synchronizations are correctly modeled. Specifically, only one thread can be inside a given critical section at a time (the others can be spinning), and threads spin on a barrier until all other threads have arrived. We compile the applications with -O3.

Workloads. We use parallel applications from PARSEC [7], SPLASH-2 [43], [6], and YCSB [13]. We evaluate 64-threaded executions of a total of 6.4B instructions or more. For the PARSEC and SPLASH-2 applications, we skip the initialization and start the evaluation when the application reaches the parallel section or the Region of Interest. We use the Medium size of the applications or larger if the trace would be too

short. We place the shared data in the CXL memory and the private data in the local memories of the CNs. As we will see, in most of these applications, the fraction of accesses that are writes to CXL memory is modest.

For YCSB, the key-value store has 500K records of size 1KB. The application utilizes a database organized in an array-like format. It issues 80% reads and 20% writes with a uniform record access distribution. In the default setup, all the key-value store data is placed in CXL and private data in the local memories. To gain insights, we also perform a sensitivity study where we move a portion of the key-value store records into the local memories, where they are replicated. In this case, the records in the local memories are not kept coherent.

IX. EVALUATION

A. Application Characterization

To understand the performance of the protocols, we start by analyzing the access patterns of the applications. Figure 12 shows the percentage of data accesses that are: reads or writes to local memory addresses, reads to remote (i.e., CXL) addresses, and writes to remote addresses. We see that remote writes are a modest fraction of all accesses. On average, they are 2% of the accesses. How the different protocols handle these potentially-expensive accesses determines the relative performance of the protocols.

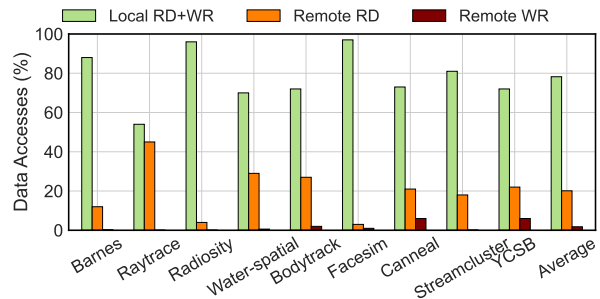


Fig. 12: Breakdown of the applications’ data accesses.

The fraction of remote writes does not depend on the data mapping. We place all the shared data in CXL memory. In these applications, most of the accesses to shared data are reads.

B. Performance

We now consider the performance of the protocols. Figure 13 compares the execution time of the applications under the five protocols considered, normalized to WB-TSO.

Consider WT-TSO first. Compared to WB-TSO, it significantly penalizes performance. It increases the geometric execution time by $2.33\times$ over WB-TSO. Large execution time increases appear even in applications such as Barnes and Raytrace that have a small fraction of remote writes (Figure 12). The reason is that most of the other accesses (local accesses and remote reads) are satisfied by cache hits with very low latency. Individual remote WT writes, however, take a very long time to complete and, therefore, determine the execution time.

Their impact on the execution time depends on how they are spaced in time. This varies across applications. Specifically, if WT writes are spread-out in time (e.g., in Streamcluster), the latency of individual WT writes can be hidden well. If, instead, WT writes occur in bunches (e.g., in Raytrace and Barnes), they can substantially slowdown WT-TSO in two ways. First, they can fill-up the SQ, causing the pipeline to stop taking new instructions. Second, if the writes occur just before a release synchronization operation (e.g., an UNLOCK), the release cannot execute until all writes complete. These are the effects that slow down WT-TSO.

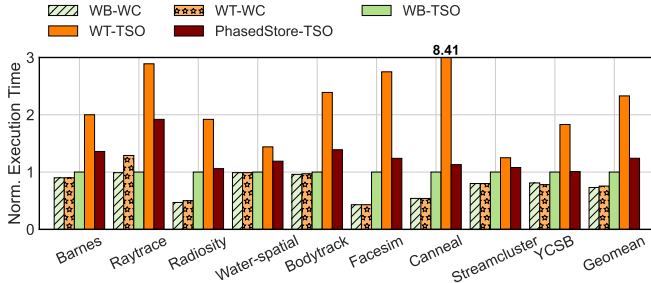


Fig. 13: Execution time of the applications for different cache coherence protocols normalized to WB-TSO.

These two effects are minimized by PhasedStore-TSO. It is true that individual WT writes to remote memory take longer to complete in PhasedStore-TSO than in WT-TSO due to the two round trips to memory needed. On average, we measure that they take $1.52\times$ longer. However, PhasedStore-TSO’s ability to *overlap* WT stores and its aggressive store *coalescing* succeed in draining the SQ faster. This speeds-up execution drastically. As a result, PhasedStore-TSO is much faster than WT-TSO. It reduces the geomean execution time by $1.88\times$.

Enforcing two round trips per write in PhasedStore-TSO also causes the directory entries to stay sealed for a longer period. However, we measure that only a geomean 0.1% of the requests reaching MNs are naked, which has minimal performance impact. This result holds for the PhasedStore-TSO system with and without coalescing.

For YCSB, we observe that PhasedStore-TSO is as fast as WB-TSO, even though YCSB has a large fraction of remote writes (Figure 12). The reason is twofold. First, because the large key-value store is accessed with a uniform record access pattern, a CN does not frequently re-write the same record, which would be a transaction that WB-TSO supports very efficiently. Second, each put operation involves writing to multiple words of the same cache line, for multiple cache lines. For this pattern, PhasedStore-TSO successfully overlaps the WT writes.

Overall, PhasedStore-TSO only takes a geomean of 24% longer than WB-TSO to execute the applications. This is a tolerable slowdown in many environments, as it comes with the ability to provide fault-tolerance guarantees through WT stores. In contrast, WB-TSO is not as resilient.

Weakly Consistent Protocols. WB-WC and WT-WC are often faster than WB-TSO because they allow all non-synchronization memory accesses to be reordered. In particular, WB-WC is $1.37\times$ faster than WB-TSO (geomean). This is a large number, which results from the fact that stores that travel to CXL memory take a long time to drain from the SQ [29]. Even WT-WC is $1.32\times$ faster than WB-TSO, showing that, with WC, we can harvest the resilience benefits of WT with minimal performance cost.

Varying the Fraction of Remote Accesses. As indicated in Section VIII, we vary the fraction of key-value store records that are stored in CXL memory; the rest are replicated in local memories. Figure 14 shows the execution time as we vary this fraction, from 100% to 0%. In the figure, the execution time is normalized to WB-TSO with a 0% fraction of remote records.

We observe that, for a 0% fraction of remote records, all TSO protocols have the same execution time. Then, as we increase the fraction of remote records, all systems become slower but, especially, WT-TSO. In all cases, PhasedStore-TSO is as fast as WB-TSO for the reasons explained above.

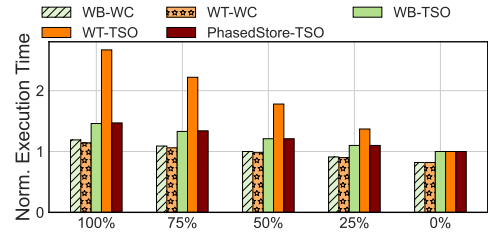


Fig. 14: Normalized execution time of YCSB when varying the fraction of records stored in CXL memory.

C. Hardware Analysis

We now analyze several hardware aspects. Figure 15a shows the fraction of cycles when the SQ is full in PhasedStore-TSO. We see that this value varies across applications. On average, it is 16.8%. The numbers on top of the bars are the reduction in total SQ-full cycles as we go from WT-TSO to PhasedStore-TSO. We see that PhasedStore-TSO reduces such cycles substantially: 62% on average. Note that an application takes a different number of cycles to execute in WT-TSO and in PhasedStore-TSO.

Figure 15b shows the fraction of remote writes coalesced for different protocols. We see that this fraction is often substantial, and increases from WB-TSO to WT-TSO and to PhasedStore-TSO. PhasedStore-TSO exhibits the highest numbers because it has aggressive coalescing. On average, it coalesces 57% of the remote stores. WT-TSO has more coalescing than WB-TSO because, in WT-TSO, writes spend more time in the SQ and, therefore, have more chances to coalesce.

Figure 15c shows the total number of messages in the CXL network divided by the total number of instructions executed, for the different coherence protocols. Messages include read and write requests and responses, invalidations,

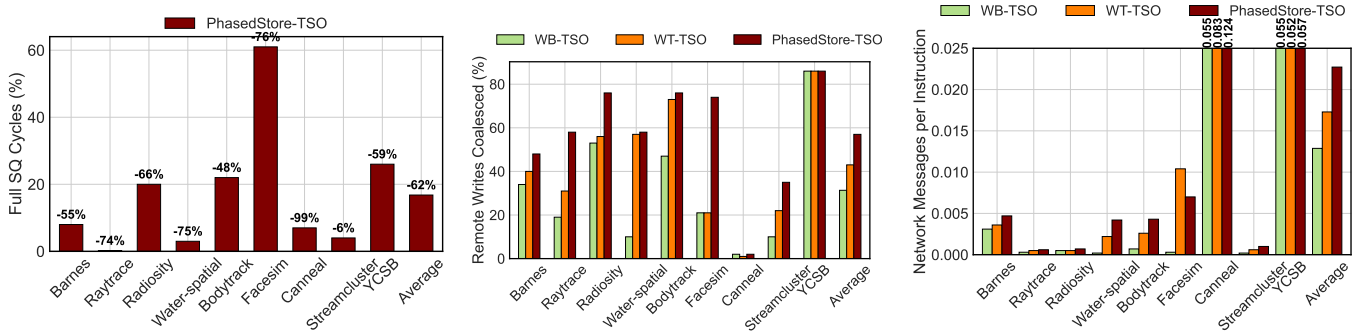


Fig. 15: Characterizing the behavior of the different coherence protocols: (a) fraction of cycles when the SQ is full; (b) fraction of remote writes coalesced; (c) CXL network messages per instruction.

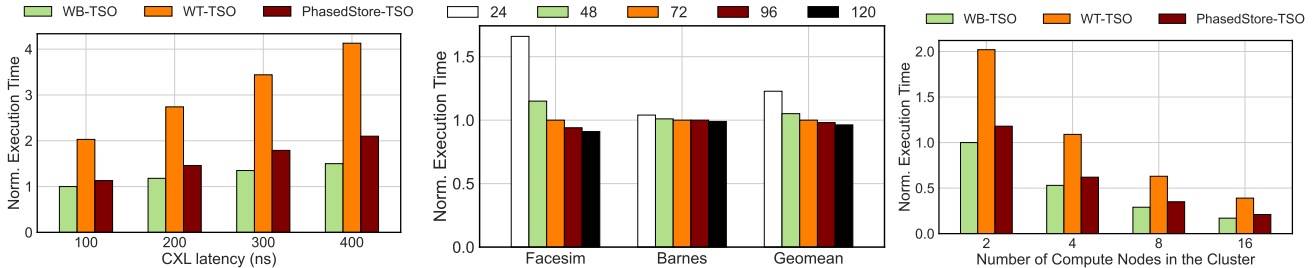


Fig. 16: Sensitivity of the coherence protocol performance to several parameters: (a) round-trip latency of the CXL network; (b) SQ size; (c) number of compute nodes (CNs).

acks, nacks, cache evictions, and prefetches. We see that the number of messages increases from WB-TSO to WT-TSO and to PhasedStore-TSO. However, the absolute numbers are all small, which means that the network bandwidth consumption is small.

D. Sensitivity Analysis

In this section, we perform several sensitivity analyses.

CXL Network Latency. Figure 16a shows the impact of the round-trip latency of the CXL network on the geomean execution time of the applications. We vary the round-trip latency from 100ns to 400ns. Our default value is 200ns. The figure shows data for different coherence protocols normalized to WB-TSO for 100ns. We see that the execution time increases for all architectures as the latency goes up. The increases are higher for WT protocols, which put more pressure on the SQ.

CXL Network Bandwidth. CXL network bandwidth is consumed by the messages measured in Figure 15c. In this experiment, we vary the bandwidth from 40GB/s to 160GB/s (our default value is 160GB/s). We do not observe significant performance changes for any coherence protocol. The primary reason is that our workloads are latency bound and not bandwidth bound.

Store Queue Size. Figure 16b shows the impact of the SQ size on execution time of the applications under PhasedStore-TSO. We vary the SQ size from 24 to 120 entries (our default value is 72). We show data for two extreme applications and for the geomean of all applications. The bars are normalized to PhasedStore-TSO for 72 entries. For some applications like

Facesim, a small SQ affects performance dramatically; for other applications like Barnes, SQ size affects performance minimally. On average, execution time increases by 23% with a 24-entry SQ and decreases by 4% with a 120-entry SQ.

Number of CNs. Figure 16c shows the execution time of the applications as we change the number of CNs from 2 to 16 (our default value is 16). Each CN has 4 cores. The bars are normalized to WB-TSO and 2 CNs. We see that adding more CNs reduces the execution time in all cases.

X. RELATED WORK

Apta [31] introduces a WT coherence protocol for fault tolerance in Function-as-a-Service (FaaS) systems using CXL memory. It is a hardware-supported transactional protocol. Each transaction reads an object, operates on it, and uses special hardware to atomically push the updated object to a single main-memory node. It supports linearizability and, therefore, conflicting transactions are squashed. The protocol introduces lazy-invalidation, which moves invalidations out of the critical path of a write, so that the writer is not blocked if a sharer fails. Special scheduling support ensures lazy invalidations do not introduce inconsistencies. In contrast, PhasedStore applies to a general-purpose environment that does not require transactions, object granularity, atomic commit, or special scheduling, and works in a cluster with many memory nodes.

Many techniques have been proposed to improve the efficiency of stores [35], [42], [37], [9], [8], [10]. They coalesce stores in the store buffer (SB) [35], utilize the L1 cache to eliminate the capacity issues of the SB [42], [8], rely on the

compiler to determine which stores can be executed out-of-order [37], and detect contiguous store access patterns in the SB and send prefetch requests for the next blocks [9]. These works are designed for WB caches.

Racer [34] is a coherence mechanism that detects data races and uses them to enforce thread ordering. In Racer, there are no directories or explicit invalidations. It has hardware structures in the LLC that detect data races between threads. When a read R from processor P_0 is found to race with a write W from processor P_1 , it is assumed that the correct program execution requires a synchronization at this point. Hence, all the updates from P_1 up to (and including) W must be merged with the LLC, P_0 must self-invalidate its private caches, and R must read from the LLC. These actions are supported by various hardware structures that include a Coalescing Store Buffer in the private caches that accumulates stores and periodically flushes them to the LLC in bulk. For multi-banked LLCs, Racer uses extensive bank-to-bank communication. Applying this to a distributed machine would require many MN-to-MN communications (which CXL does not support). This specialized scheme supports TSO.

Prior designs coalesce WB stores in the SQ after they retire [42], [35], [10], [8]. To enable the coalescing of stores under TSO, some methods rely on speculation [42]. Ros and Kaxiras [35] propose instead that stores coalesce into atomic groups, where stores in a group can be issued out-of-order. Their approach requires locking the cache lines targeted by the stores in the group in lexicographical order, in a sequential manner. Applying their approach to a WT environment would require the protocol to have an "exclusive" state just like WB caches. This would add extra hardware complexity to WT protocols. Moreover, they grab the locks for the cache lines serially, which induces high overhead if the cache lines are not already in the exclusive state in the caches. Other designs combine stores in write buffers to improve write-update protocols [14] or relaxed memory consistency models [17].

XI. CONCLUSION

This paper presented PhasedStore, the first architecture that supports high-performance cache-coherent WT stores under TSO. PhasedStore allows WT stores to be fully overlapped and still satisfy TSO. We evaluate PhasedStore with simulations of a 16-CN CXL cluster running parallel applications. Compared to a conventional WT implementation, PhasedStore reduces the average execution time of the applications by 1.88x.

ACKNOWLEDGMENTS

We thank the reviewers for improving this paper. This work was supported by NSF with grants CCF 2107470 and CCF 2316233, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. J. Marathe, A. Xygiak, and I. Zabolotchi, "Microsecond Consensus for Microsecond Applications," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, 2020.

[2] C. Alverti, V. Karakostas, N. Kunati, G. Goumas, and M. Swift, "DaxVM: Stressing the Limits of Memory as a File Interface," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022. [Online]. Available: <https://doi.org/10.1109/MICRO56248.2022.00037>

[3] AMD, "AMD Ryzen 9 9950X3D Processor," 2025. [Online]. Available: <https://www.amd.com/en/products/processors/desktops/ryzen/9000-series/amd-ryzen-9-9950x3d.html>

[4] Arm Ltd., *Arm AMBA CHI Architecture Specification*, 2024. [Online]. Available: <https://developer.arm.com/documentation/ih0050/latest>

[5] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming," HP Laboratories, Tech. Rep., 2012.

[6] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, D. Christie, A. Lee, O. Mutlu, and B. G. Zorn, Eds. IEEE Computer Society, 2008, pp. 47–56. [Online]. Available: <https://doi.org/10.1109/IISWC.2008.4636090>

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[8] J. M. Cebrian, M. Jahre, and A. Ros, "Temporarily Unauthorized Stores: Write First, Ask for Permission Later," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 810–822.

[9] J. M. Cebrian, S. Kaxiras, and A. Ros, "Boosting store buffer efficiency with store-prefetch bursts," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 568–580.

[10] Y. Chou, L. Spracklen, and S. G. Abraham, "Store memory-level parallelism optimizations for commercial applications," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE, 2005, pp. 12–pp.

[11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011.

[12] Compute Express Link Consortium, "CXL Specification," 2024. [Online]. Available: <https://computeexpresslink.org/cxl-specification/>

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: <https://doi.org/10.1145/1807128.1807152>

[14] F. Dahlgren, M. Dubois, and P. Stenström, "Combined performance gains of simple cache protocol extensions," *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2, pp. 187–197, 1994.

[15] D. Das Sharma, R. Blankenship, and D. Berger, "An Introduction to the Compute Express Link (CXL) Interconnect," *ACM Computing Surveys (CSUR)*, 2023. [Online]. Available: <https://doi.org/10.1145/3669900>

[16] D. L. Dill, "The Mur ϕ verification system," in *International Conference on Computer Aided Verification*. Springer, 1996, pp. 390–393.

[17] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen, "Delayed consistency and its effects on the miss rate of parallel programs," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 197–206.

[18] K. Gharachorloo, A. Gupta, and J. L. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," in *Proceedings of the 1991 International Conference on Parallel Processing (ICPP)*, vol. 1, 1991, pp. 355–364.

[19] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *International Symposium on Computer Architecture (ISCA)*, 1990.

[20] B. A. Hechtman, S. Che, D. R. Hower, Y. Tian, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "QuickRelease: A throughput-oriented approach to release consistency on GPUs," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 189–200.

- [21] Y. Huang, H. Chen, N. Ni, Y. Sun, V. Chidambaram, D. Tang, and E. Witchel, "Tigon: A Distributed Database for a CXL Pod," in *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*, Boston, MA, USA, 2025.
- [22] P. Hunt, M. Konar, F. P. Junqueira, and B. C. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, 2010.
- [23] Intel, "Pin - a dynamic binary instrumentation tool," 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html>
- [24] Intel, "Intel Core i9-14901TE Processor (36M Cache, up to 5.50 GHz)," 2025. [Online]. Available: <https://www.intel.com/content/www/us/en/products/sku/238778/intel-core-i9-processor-14901te-36m-cache-up-to-5-50-ghz/specifications.html>
- [25] Intel Corporation, "eADR: New Opportunities for Persistent Memory Applications," 2021, accessed: 2025-04-10. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html>
- [26] Intel Corporation, "Intel Sapphire Rapids," 2023. [Online]. Available: <https://www.intel.com/content/www/us/en/newsroom/news/4th-gen-xeon-scalable-processors-max-series-cpus-gpus.html>
- [27] S. Jain, N. Yelleswarapu, H. A. Maruf, and R. Gupta, "Memory Sharing with CXL: Hardware and Software Design Approaches," *arXiv preprint arXiv:2404.03245*, 2024. [Online]. Available: <https://arxiv.org/abs/2404.03245>
- [28] S. Kumar, A. Shriraman, and N. Vedula, "Fusion: Design tradeoffs in coherent cache hierarchies for accelerators," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 733–745.
- [29] J. Liu, H. Hadian, Y. Wang, D. S. Berger, M. Nguyen, X. Jian, S. H. Noh, and H. Li, "Systematic CXL Memory Characterization and Performance Analysis at Scale," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25, 2025. [Online]. Available: <https://doi.org/10.1145/3676641.3715987>
- [30] Micron Corporation, "CXL Memory Expansion: A Closer Look on Actual Platform," 2024. [Online]. Available: <https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansion-a-close-look-on-actual-platform.pdf>
- [31] A. Patil, V. Nagarajan, N. Nikoleris, and N. Oswald, "Apta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS," in *Proceedings of the 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '23)*, 2023. [Online]. Available: <https://doi.org/10.1109/DSN58367.2023.00030>
- [32] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous system coherence for integrated CPU-GPU systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 457–467.
- [33] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "The Structural Simulation Toolkit," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 4, pp. 37–42, 2011.
- [34] A. Ros and S. Kaxiras, "Racer: TSO consistency via race detection," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.
- [35] A. Ros and S. Kaxiras, "Non-speculative Store Coalescing in Total Store Order," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18, 2018. [Online]. Available: <https://doi.org/10.1109/ISCA.2018.00028>
- [36] 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," *CACM*, no. 7, pp. 89–97, Jul. 2010.
- [37] S. Singh, A. Jimborean, and A. Ros, "Regional Out-of-order Writes in Total Store Order," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, 2020, pp. 205–216.
- [38] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, "Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023. [Online]. Available: <https://doi.org/10.1145/3613424.3614256>
- [39] J. Terrace and M. J. Freedman, "Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads," in *Proceedings of the 2009 USENIX Annual Technical Conference, USENIX ATC 2009, San Diego, CA, USA, June 14-19, 2009*, G. M. Voelker and A. Wolman, Eds. USENIX Association, 2009.
- [40] R. van Renesse and F. B. Schneider, "Chain Replication for Supporting High Throughput and Availability," in *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. USENIX Association, 2004, pp. 91–104.
- [41] D. L. Weaver and T. Germond, *The SPARC architecture manual, Version 9*. Prentice-Hall, 1994.
- [42] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 266–277.
- [43] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*, D. A. Patterson, Ed. ACM, 1995, pp. 24–36. [Online]. Available: <https://doi.org/10.1145/223982.223990>
- [44] Y. Xu, S. Mahar, Z. Liu, M. Shen, and S. Swanson, "CXL Shared Memory Programming: Barely Distributed and Almost Persistent," *arXiv preprint arXiv:2405.19626*, 2024.
- [45] X. Yang, Y. Zhang, H. Chen, F. Li, G. Fan, Y. Kong, B. Wang, J. Fang, Y. Wang, T. Huang, W. Hu, J. Kao, and J. Jiang, "Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases," in *Companion of the 2025 International Conference on Management of Data*, ser. SIGMOD/PODS '25, 2025.
- [46] Y. Yu, N. Oswald, and A. Khandelwal, "CORD: Low-Latency, Bandwidth-Efficient and Scalable Release Consistency via Directory Ordering," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 1311–1326.
- [47] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu, "Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23, 2023.
- [48] Z. Zhu, N. Ni, Y. Huang, Y. Sun, Z. Jia, N. S. Kim, and E. Witchel, "Lupin: Tolerating Partial Failures in a CXL Pod," in *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, ser. DIMES '24, 2024. [Online]. Available: <https://doi.org/10.1145/3698783.3699377>