2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2235-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/HPCA51647.2021.00021

TSOPER: Efficient Coherence-Based Strict Persistency

Per Ekemark*§, Yuan Yao*§, Alberto Ros†, Konstantinos Sagonas*‡, and Stefanos Kaxiras*

*Dept. of Information Technology
Uppsala University
Uppsala, Sweden

†Computer Engineering Dept.
University of Murcia
Murcia, Spain

‡School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece

*{per.ekemark,yuan.yao,konstantinos.sagonas,stefanos.kaxiras}@it.uu.se †aros@ditec.um.es

***Abstract*—We propose a novel approach for hardware-based strict TSO persistency, called TSOPER. We allow a TSO persistency model to freely coalesce values in the caches, by forming atomic groups of cachelines to be persisted. A group persist is initiated for an atomic group if any of its newly written values are exposed to the outside world. A key difference with prior work is that our architecture is based on the concept of a TSO persist buffer, that sits in parallel to the shared LLC, and persists atomic groups directly from private caches to NVM, bypassing the coherence serialization of the LLC.**

To impose dependencies among atomic groups that are persisted from the private caches to the TSO persist buffer, we introduce a sharing-list coherence protocol that naturally captures the order of coherence operations in its sharing lists, and thus can reconstruct the dependencies among different atomic groups entirely at the private cache level without involving the shared LLC. The combination of the sharing-list coherence and the TSO persist buffer allows persist operations and writes to non-volatile memory to happen in the background and trail the coherence operations. Coherence runs ahead at full speed; persistency follows belatedly.

Our evaluation shows that TSOPER provides the same level of reordering as a program-driven relaxed model, hence, approximately the same level of performance, albeit without needing the programmer or compiler to be concerned about false sharing, data-race-free semantics, etc., and guaranteeing all software that can run on top of TSO, automatically persists in TSO.

I. Introduction

Non-volatile memory (NVM) has attracted significant attention both as a technology [2], [24], [27], [28], [34] and as a novel architectural feature [11]. NVM introduces a new challenge in the hardware/software interface. The question that arises is: what is the observed order in which memory writes are *persisted* in NVM? This is similar to the question that memory consistency aims to address, albeit with an important difference. While consistency deals with the order in which memory accesses appear to be performed in memory (memory order) in relation to the order they appear in the program (program order), ***persistency*** addresses the question of what is the observed order of persists in NVM *after a crash*, for persist operations that occurred prior to the crash.

The seminal work of Pelley et al. [31] introduced the term *persistency* and identified *strict* and *relaxed* classes of persistency models. Strict persistency semantics adhere to the underlying memory consistency model: the order of stores, as seen by observers in the consistency model, is preserved in the persist order. As the authors note [31], conceptually, the same mechanisms provided by the consistency model to enforce store order can be used to enforce persist order. However, this proves to be expensive. Relaxed persistency semantics, on the other hand, decouples persistency order from consistency order and allows the order of persist operations to deviate from the order in which the corresponding stores become visible in the consistency model. An example of a relaxed persistency model proposed by Pelley et al. is epoch persistency [31], which eventually led to language-level persistency models [8], [16], [25] and persistency for synchronization-free regions [15].

Relaxed persistency models are potentially a very good fit for relaxed consistency models as both require programmer involvement for correctness. In other words, the programmer has to fence both for the consistency model *and* for the persistency model, and the effort for the former may be leveraged for the latter. However, in architectures that implement a stronger consistency model, such as x86-TSO, we are faced with a discrepancy between the consistency model and the persistency model. This discrepancy exists today between x86-TSO and the relaxed persistency model introduced by Intel [19] and formally described, as Px86, by Raad et al. [33] The consistency model, x86-TSO, requires little or no involvement from the programmer. In contrast, the persistency model, Px86, requires significant involvement with insertion of `CLFLUSH` (program-ordered, buffered, persist operation), `CLFLUSHOPT` (unordered, buffered, persist operation), `CLWB` (unordered, buffered, persist operation), and `S/MFENCE` (persist barrier), in the proper places in the code. This involvement is comparable to the effort that would be needed for a relaxed consistency model. This largely negates the benefit of having TSO to begin with, as anyone who wishes to achieve correct persistency would have to fence the programs as if they were meant to run on relaxed consistency. This is problematic for all existing software that runs on x86-TSO but is not fenced for persistency.

Furthermore, relaxed models (either for consistency or persistency) rely on *data-race-free* (*DRF*) semantics for correctness. However, the burden falls on the programmer to provide DRF guarantees. Arguably, correct, well-behaved programs should adhere to DRF semantics —a stance also reflected in the definition of the memory models of modern languages

This work was supported in part from the EU Horizon 2020 research and innovation programme (grant No 801051), from the European Research Council (ERC grant No 819134), and from Vetenskapsrådet (grant No 2018-05254).
§First authors ordered alphabetically.

978-1-6654-2235-2/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00021

(e.g., C++11 [5])— and data races should be considered *bugs*. Unfortunately, this reasoning has two issues: (1) it does not take into account legacy software that is written for TSO and which may have data races as optimizations to synchronization [39]; and (2) it does not take into account data races at a *coarser granularity* than individual variables, i.e., data races that are due to *false sharing*.

In contrast, a *hardware-only* implementation of a strict persistency model that adheres to a consistency model such as TSO (or SC) is not plagued by such problems, because it can simply detect run-time conflicts at cacheline granularity and react accordingly. Thus, without any software involvement, a program compatible with TSO will also persist correctly on hardware-only strict TSO persistency. Furthermore, any DRF program with no additional annotations that runs correctly on TSO also persists correctly on this persistency architecture.

Proposal: Our work provides a new solution for an efficient *hardware-only* implementation of a strict TSO persistency model, called *TSOPER*. TSOPER relies on a TSO persist buffer [32] that sits in parallel to the LLC. Private caches persist directly to this buffer bypassing the coherence serialization imposed by the shared LLC. This is our main differentiation point from the prior state-of-the-art, Buffered Strict Persistency (BSP) [22], and the driver for our design decisions.

Our insight is that we can use coherence to both automatically create the proper "epochs" per thread in the corresponding private caches *and* order these epochs according to the data dependencies (among threads) that arise at runtime. The term "epoch," however, refers to program (instruction) execution. We fully decouple persistency from program execution as our approach is *not* software-driven. Thus, in our work, we talk about *atomic groups of cachelines (AGs)*, rather than epochs, but we note that there is a relation between the two.

An atomic group is created and expands to include locally-modified cachelines in the private cache of a thread as long as no local modification is exposed to other threads (either via external requests or via cache or directory evictions). Inspired by BulkSC [7], our atomic groups preserve TSO, but in contrast to BulkSC, we do so *non-speculatively*. If the cachelines of an atomic group are persisted *atomically* (with no intervening conflicting persist), we maintain TSO persistency regardless of the order the individual cachelines of the group are persisted. Interactions with the outside world, concerning the cachelines of an atomic group, either create dependencies for the atomic group, when it sees the modifications of other atomic groups, or freeze the atomic group, when it is forced to expose its own modifications to other atomic groups. Freezing an atomic group automatically starts the process of persisting it; a new atomic group (in program order) starts forming in the private cache of the same thread, capturing subsequent stores for this thread. Similarly to other approaches (e.g., BSP [22] and SFR decoupled [31]), the dependencies among atomic groups (epochs in other approaches) must be respected in the persist order of the atomic groups. *The key insight of this work is that dependencies among atomic groups can be fully captured at the coherence level, and in particular entirely at the L1 caches.*

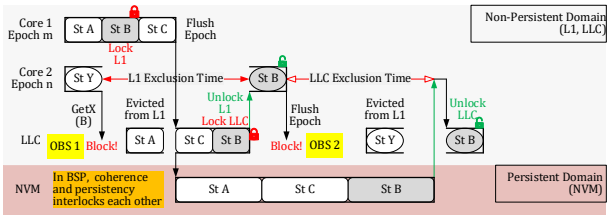
To demonstrate this capability, we develop a sharing-list protocol, inspired by SCI [20], that naturally captures in a sharing list the order in which the coherence operations for a block are serialized by the directory. In such a protocol, different writers of a shared block queue up one after another in the list, and perform their persists in the order in which their stores were inserted in the memory order. Persistency is enforced belatedly, trailing coherence, but following the same order. A block's sharing list is dismantled only by the ordered persist of the locally-modified cachelines on the sharing list.

By ensuring that it is impossible to create dependence cycles among atomic groups, we guarantee *deadlock-free* TSO persistency with enough flexibility to coalesce multiple stores in cachelines and re-order individual cacheline persists (within an atomic group) to match the level of performance of the relaxed persistency models. We do not rely on any kind of speculation or transactional approach that would detect conflicts, roll-back, and retry. Our approach is strictly *non-speculative*, rendering any cost related to out-of-core speculation (checkpoints, maintaining speculative state, commit overhead, etc.) unnecessary.

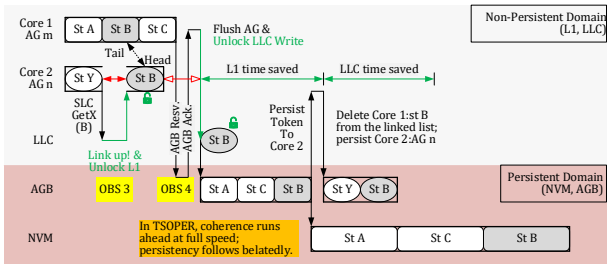
We envision a unified mechanism that enforces *coherence* to support the consistency model (TSO) *and* enforces the *persistency* model. For simplicity, we use the *same* sharing-list protocol for both coherence and persistency in this paper, but, in principle, a different protocol could be used for coherence. Using the same sharing-list protocol for both coherence and persistency has the advantage of easily demonstrating the decoupling of coherence and persistency: coherence happens at the heads of the sharing lists (the young memory order end) while persistency happens at the tails of the sharing lists (the old memory order end).

As in other persistency models, including Px86, SFR persistency, and PTSO, we also employ *persist buffering* to decouple *program execution* from the actual persist operations that eventually reach the NVM. In our model, a TSO persist buffer, similarly to Intel's *Write Pending Queue* [37], guarantees that atomic groups, persisted directly from the L1 caches, will be made durable in NVM even in the event of a crash (e.g., power failure). Our TSO persist buffer, called *Atomic Group Buffer (AGB)*, accommodates multiple versions of same-address cachelines in different atomic groups awaiting their writing to NVM. This is a fundamental differentiation point from BSP (and LAD [18]) that persist through the LLC and impose a single-version restriction: *visibility* and writing to NVM must be interlocked on the same version.

While the AGB is designed to accommodate multiple atomic groups in flight to NVM, its storage requirements are especially frugal, ensuring that it is feasible with existing technology. In our evaluation (§V), we model an AGB per memory channel of modest size (10 KB) that can fit two large atomic groups (of up to 80 cachelines). Atomic groups are generally small (90% of the atomic groups that are formed in the evaluated workloads are under 10 cachelines—see §V) and, thus, the AGB size can be easily reduced to one eighth (i.e., 1.25 KB per memory channel) without significantly impacting performance. This



(a) BSP.



(b) TSOPER.

Fig. 1: Comparison between BSP [22] and TSOPER.

places our design squarely at the level of *current, commercially available, WPQ implementations*, which are evaluated by Wang et al. [45].

Advancements beyond the State of the Art: Figure 1 depicts the differences of TSOPER over the state-of-the-art, BSP. Enabled by our techniques, TSOPER aims to hide the overhead of persisting to NVM. From the figure, we can make the following observations:

OBS 1. L1 exclusion time. In BSP, St B in {Core 1:Epoch m} locks cacheline B in L1, preventing invalidations from remote cores until St B is written to LLC. This protects Core 1 from losing the value of St B before Epoch m is persisted so that epoch atomicity is preserved. As shown in Fig. 1a, St B in {Core 2:Epoch n} is blocked from updating its local L1 until St B in {Core 1:Epoch m} is installed in the LLC.

Definition 1: We define the time from the moment a core issues a write request (GetX) until the moment the write permission is granted as the **L1 exclusion time** *(see Fig. 1a).*

Moreover, because cachelines within the same epoch are persisted without any particular order, *worst case* L1 exclusion time is a function of epoch size (i.e., when the cacheline in question is written *last* to the LLC), as shown in Fig. 1a.

OBS 2. LLC exclusion time. In BSP, the LLC acts as the landing point between L1 and NVM, i.e., every epoch should first be written to LLC then to NVM. Thus, the LLC can accept a newer-epoch version of a cacheline from L1 only after older-epoch version of the same cacheline in the LLC is persisted in NVM. In Fig. 1a, after St B from {Core 1:Epoch m} reaches the LLC, it "locks" cacheline B in the LLC until St B has persisted to NVM. During this time, St B from Core 2 is restricted from updating the LLC.

Definition 2: We define the time from the moment a core issues an epoch flush request to the LLC until the moment the epoch flush request is granted as the **LLC exclusion time** *(Fig. 1a).*

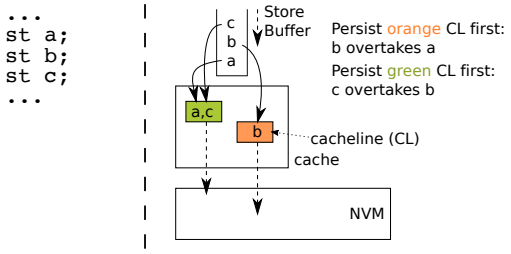


Fig. 2: Store coalescing in cachelines violates TSO persistency: persisting the green cache block will violate the TSO between `b` and `c`; persisting the orange cache block will violate the TSO between `a` and `b`. There is no way that cachelines can be persisted that does not violate TSO.

As can be observed, LLC exclusion time is jointly determined by the epoch size and the NVM write latency, which is typically hundreds of cycles per write.

OBS 3. ***Our first innovation:*** **Reducing the L1 exclusion time using the SLC protocol.** In TSOPER, a sharing list protocol (SLC, see §IV) allows multiple writers to co-exist in the sharing list, eliminating the need to lock in the L1 cache. Different writers of a shared block queue up one after another in the sharing list, and perform their persists in the order in which their stores were inserted in the memory order. In this way, Core 2 can install St B in the private cache immediately after the cacheline is linked up in the sharing list,[1] significantly reducing the L1 exclusion time.

OBS 4. ***Our second innovation:*** **Reducing the LLC exclusion time using AGB.** In TSOPER, atomic groups are directly persisted from L1s to the AGB, in parallel to being written in the LLC, which nearly eliminates the LLC exclusion time. Similarly to Intel's *Write Pending Queue* [37], the AGB guarantees that atomic groups, persisted directly from the L1 caches, will be made durable in NVM even in the event of a power failure or crash. To reduce the high write latency to NVM, AGB banks are implemented with faster technology such as battery-backed SRAM. More details on the AGB organization are in §II-A.

Finally, we show through detailed simulation that our approach decouples coherence and persistence further than the state of the art prior solution, Buffered Strict Persistence [22], and achieves performance levels close to relaxed persistency (3% performance overhead on the average).

II. TSO Persistency

TSOPER is an efficient, strict TSO persistency model that relies on coherence and TSO persist buffering for its implementation. Compared to both PTSO [32] and Intel's Px86 model (as formally described by Raad et al. [33]), we are making TSOPER *transparent to the software* as far as the *persist ordering* of the stores is concerned.

Under strict persistency, persist order coincides with consistency order. For TSO, this means that persists must follow the store order as prescribed by program order. Unfortunately, this introduces many dependencies. Refer to Fig. 2. In this

[1]A new writer is inserted as the new "head" in a doubly-linked sharing list.

example, a thread executes three stores to a, b, and c, in this order. Two of these stores, st a and st c, coalesce in the same cacheline with the third store, st b, interposing between them. Strict TSO persistency requires that *three* separate persist operations be performed, one for each store.

If we attempt to capitalize on *coalescing* to reduce the persists down to two (one for the green cacheline and one for the orange one) and a crash occurs between the persists, persistent memory will be updated incorrectly with respect to TSO: Persisting the green cacheline violates the store order between b and c; persisting the orange cacheline violates the order between a and b. There is no persist order for the cachelines that does not violate TSO in a possible crash between the persists.

To solve this problem, PTSO integrates (buffered) epoch persistency [31] with the TSO consistency model [41]. Under epoch persistency, the execution of each thread is delineated into persistency epochs by explicit, programmer-inserted, pfence instructions. Stores within an epoch can persist in any order, preserving only the order of the stores to the same address. A pfence instruction acts as a *persist barrier*, ensuring that no stores following the barrier are persisted before the stores preceding the barrier. A persist barrier is defined by Pelley et al. as CLWB; SFENCE for Intel's Px86.

Persist buffering, in turn, decouples the execution of the pfence (or SFENCE) instruction from the actual completion of the individual persists in NVM. Such persist buffering already exists in actual processors, e.g., Intel's Write Pending Queue (WPQ) [37]. As with the WPQ, we assume that buffered persists are considered committed to NVM even in the event of a crash. However, we expand the WPQ concept to a TSO persist buffer that operates at the atomic group level and is visible by loads. Section III expounds on the AGB functionality.

While PTSO prescribes how to create epochs (by inserting pfence instructions in a program) that guarantee TSO persistency, TSOPER *automatically creates* the *analogous* of epochs by *observing and reacting to coherence transactions that denote data races (conflicts)*. We say the "analogous of epochs" because we do not intervene in program execution to create epochs; we work at the cache level, decoupled from program execution. In contrast, epochs in Joshi et al. [22] are delineated by software–inserted (BEP) or hardware-inserted persist barriers (BSP). Our approach is built on the concept of an *atomic group (AG) of cachelines*. Broadly speaking, an AG incorporates the changes made in the private cache during a period between two successive exposures of local data modifications to the outside world. In this perspective, an atomic group is not directly connected to a program epoch.

A. From Epochs to Atomic Groups

In TSOPER, as the three stores of the example in Fig. 2 exit the store buffer and write to the cache, an atomic group is created and expands to contain both the orange ([a,c]) and the green ([b]) cacheline (Fig. 3). There is no immediate need to persist this atomic group (as opposed to hitting a pfence in PTSO) but if any of the two cachelines (green or orange)

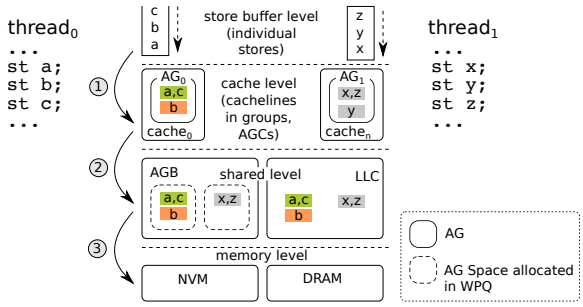


Fig. 3: System model: (1) Stores a, b, c coalesce in cachelines [a,c] and [b] and form atomic group AG_0; (2) AG_0 cachelines are buffered in the Atomic Group Buffer (AGB); (3) when complete in AGB, AG_0 is made durable in NVM. In contrast, AG_1 is not fully persisted in AGB, therefore, none of its cachelines are made durable in NVM.

were to be exposed to the outside world, the atomic group would have to be persisted on such exposure. Of course, as Fig. 2 shows, there is no persist order for individual cachelines that would maintain TSO (if we have coalesced stores). This is why we need an atomic group to persist *atomically*, i.e., *all or nothing and without any intervening conflicting persist*. Same as with persist epochs in PTSO, an atomic-group persist maintains TSO for the stores it encapsulates, no matter what the persist order of individual cachelines is.

To summarize from a different perspective: Stores exit a FIFO store buffer (i.e., in program order) and locally modify cachelines in a cache. As long as the cache is not forced to reveal any of its modifications to the outside world, it can continue to accept stores without having to persist anything. At the point, however, that the cache must reveal any of its locally-modified cachelines, either because of an eviction or because of a coherence request (someone else wishes to read or write a locally-modified cacheline), the cache must go into a persist mode and persist its dirty cachelines as an AG.

There are two observations, here, that give us a greater insight into the formation of atomic groups:

1. *No other cache in the system has a (valid) copy of a cache's locally-modified cachelines.* We assume single-writer/multiple-reader semantics (SWMR), and the cache is the *exclusive* holder of its locally-modified cachelines.
2. The set of locally-modified cachelines in a cache is effectively *locked in place* simply by means of cacheline exclusivity: the release of these cachelines to the outside world, is wholly controlled by the cache that holds them.

Atomicity, for persisting an atomic group, is implemented via a persist buffer which fills a similar role of a logging mechanism. This buffer must guarantee that we are not left with a *partial* atomic group in NVM, when a crash occurs in the middle of making an atomic group durable.

The Basics: Creating, Freezing, and Persisting AGs: AGs are formed in private caches. An AG, generally, contains the locally-modifed cachelines —but as we will see it can also contain locally-unmodified cachelines that are read from other AGs— up to the point where it is forced to expose its modifications.

This happens when:

1. An eviction of a locally-modified cacheline forces a write-back to a shared cache level (or memory).
2. A directory eviction forces the writeback of a locally-modified cacheline to memory.
3. Another core requests a downgrade/invalidation, wanting to read/write a locally-modified cacheline via a coherent read/write request.
4. The AG reaches the size of the persist buffer. AGs cannot exceed this size limit; otherwise, their atomicity in NVM is not guaranteed.

On such an event, the AG is *frozen*, i.e., it can no longer expand, and must be persisted to NVM. We will detail this procedure in later sections.

When we decouple coherence from persistency, more than one AG can be present in a private cache at any one time. In that case, AGs are ordered in *program order*: the oldest AG is the one to persist first, followed by other AGs, and with the youngest AG being the one that is currently expanding to accommodate stores exiting the store buffer.

An atomic group ID, *AG_ID*, tags the cachelines of an AG. In practice, only few AGs are present in a cache at any point in time, thus, only a few bits are needed for the AG_ID. Two AG_ID registers indicate the oldest (first to persist) and the youngest (still open) AG in the cache. A stall will occur if after AG_ID wraparound, the youngest ID catches up with the ID of the oldest group in the cache, but the size of the IDs can be adapted to ensure this rarely happens. AG_IDs are locally managed by L1 controllers, i.e., each core assigns its own sequence of AG_IDs to its local AGs. There is no global AG_ID number, thus no synchronization on AG_ID is necessary.

Multiversioning: A store, trying to exit the store buffer and write to the local cache, is blocked if tries to write a cacheline in a *frozen* atomic group. We do, however, allow multiple versions of the same block to reside across *different* caches for the purpose of persistency as we explain in §IV. All except one of these versions are invalid and part of a frozen AG. The only usable valid version is owned by the last writer and is part of an open AG. We use a sharing list protocol to maintain the order in which these versions must be persisted (§IV).

Atomicity: Atomicity of an AG is guaranteed by two mechanisms: The first mechanism is to ensure that atomic groups are made durable in NVM *either in their entirety or not at all*. This is accomplished by the atomic group persist buffering mechanism, shown in Fig. 3 and explained in the following sections. The second mechanism to guarantee AG atomicity concerns the ordering constraints that we place on the persist order of the atomic groups so as to *order conflicting or dependent persists* and guarantee TSO. In §III, we explain how we establish the *dependencies* among atomic groups and how we order them for persist without deadlock.

B. Persisting via the Atomic Group Buffer

The key advancement of our work is that our system architecture is based on the concept of a FIFO TSO persist buffer that persists atomic groups *directly from L1s to NVM*. The TSO persist buffer is exemplified in the Px86 model of Raad et al. [32]. In our work, the TSO persist buffer is the Atomic Group Buffer (AGB), that sits in parallel to the LLC. In contrast, related works (e.g., BSP [22] and LAD [18]) persist through the LLC, which imposes single-value semantics.

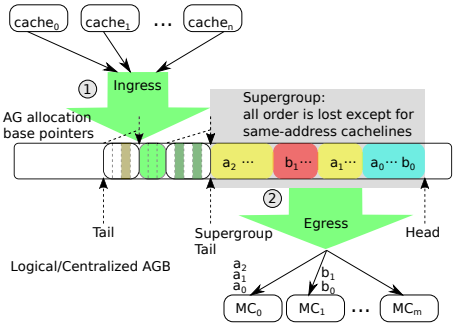


Fig. 4: Logical (centralized) view of AGB.

This architectural choice provides a new degree of decoupling between persistence and coherence. In BSP, the LLC can accept a newer-epoch version of a cacheline *only* after the LLC's older-epoch version of the same cacheline is *persisted* in NVM. In TSOPER, the LLC is constantly updated with the newest-epoch version of a cacheline while simultaneously enqueueing the same version in the AGB. The durable write from the AGB to NVM is thus fully decoupled from the LLC.

Similarly to Intel's WPQ, the AGB is in the persistent domain. It guarantees the durability of its contents in NVM even in the event of a crash. To *persist* an AG, we simply buffer it cacheline-by-cacheline from the L1 to the AGB. To guarantee its durability in NVM, we *wait for the whole AG to be placed in the AGB* before making any of its cachelines durable in NVM. The only limitation imposed by this approach is that an AG cannot exceed the size of the AGB. If an AG cannot fit, because other AGs are buffered in the AGB, we stall its buffering until enough space is available for it to fit. In case of a crash, the AGB will not make durable in NVM any AG that is not completely buffered, as previously shown in Fig. 3.

It is important to note that because the AGB works at the atomic-group level, all order is relaxed for the cachelines within an atomic group; conceptually, FIFO semantics apply to AGs but as we will show below this degenerates to FIFO semantics applying only for ***same-address cachelines***.

AGB Ingress: Figure 4 provides an overview of the logical structure of the AGB. The goal of the AGB ingress (Fig. 4 ①) is to allow concurrent AGs (persisting in parallel from different L1s) to be buffered easily. To avoid complex scheduling in the AGB when we persist concurrent AGs, we simply reserve space for the whole atomic group when we buffer its first cacheline.[2] For each AG, cachelines are allocated consecutively in the AGB, starting from an *allocation base pointer* that is assigned

[2]An AG's size is known by the time it is frozen, which happens before its first cacheline is persisted.

to the AG on allocation. AGs, in turn, are laid out consecutively, on a *first-come first-served* basis, if they are concurrent. If they have dependencies, the dependency ordering mechanism (§III) orders their allocation. After allocating an AG, the L1 sends its AG cachelines (in any order) to the AGB *and* the LLC.

AGB Egress: While AG ingress in the AGB is ordered, AG *egress is largely unordered* and provides ample parallelism for multiple memory controllers (Fig. 4 ②). Consecutive, fully-persisted AGs, starting with the AG at the head of the AGB, form an ***atomic super group*** that is guaranteed to become durable in NVM. Thus, the order among unique cachelines within this super group *does not matter*. On the other hand, the order among *same-address* cachelines (e.g., a0, a1, a2 in Fig. 4) within this super group matters: same-address cachelines are written in NVM in a FIFO fashion. In other words, AG dependencies are implicitly encoded in the AGB in the allocation order of same-address cachelines. Same-address order is automatically taken care of by the mapping of addresses to memory controllers (MCs).

Searching the AGB: Like any TSO store buffer, loads must take the latest value from AGB before going to NVM. Since the LLC is always updated with the latest version of the data in parallel to the AGB, the latter needs to be searched only on LLC misses. We can easily afford to serially search a small-sized AGB (as the one used in §V), under the shadow of an LLC miss. Further, we can completely avoid searching the AGB by making the LLC always *inclusive* of the contents of the AGB, e.g., by *pinning* LLC cachelines, from the time they enter the AGB to the time they exit the AGB. We leave this as a future optimization.

C. AGB Organizations

Centralized AGB: In this model (Fig. 4), the non-volatile state of the MCs is unified in a (power-backed) centralized, circular, SRAM buffer (AGB). The AGB is written by multiple private caches in parallel and drained out to multiple MCs. Unique cachelines in the AGB super group can be written out in any order to the corresponding MCs. Same-address cachelines are routed to the same MC. The AGB is scanned head to super-group tail and can skip over cachelines, but it never reorders cachelines going to the same MC.

Distributed AGB: The AGB itself can be distributed across multiple memory controllers as shown in Fig. 5. In that case, the cachelines of an AG are distributed to the corresponding AGB slice based on their address. Slices of an AG are allocated to AGB slices in a single step via a centralized *AG allocation arbiter* (Fig. 5 ①). This arbitration is usually not in the critical path as it concerns the ingress of the AGB slices and therefore is often hidden behind fully-persisted AGs awaiting to be written to NVM. Once an AG slice is filled in an AGB slice, the arbiter is signalled (Fig. 5 ②) which in turn broadcasts to all AGB slices (Fig. 5 ③) when the AG is fully entered everywhere. After this two-phase allocation-completion ingress the AG slices become part of the supergroups in the corresponding AGB slices, and, in contrast to BSP [22], no further coordination is needed for the egress (Fig. 5 ④). This

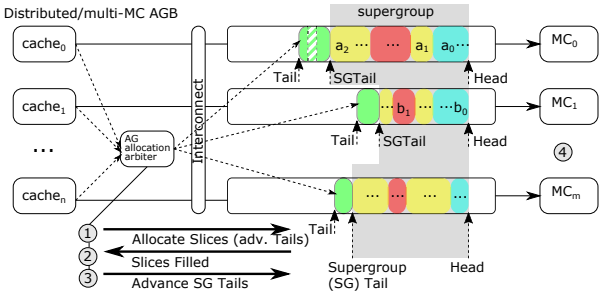


Fig. 5: On-chip distributed AGB (multiple MCs).

is because cachelines within a super group can be written out to NVM in any order regardless of their AG order. Of course, *same-address* cachelines are written in FIFO order by virtue of being enqueued in the same AGB slice (same MC) based on their address. This property does not hold in cross-chip distribution where each chip has its own set of MCs, and *MC exclusivity in handling address sets is lost*. Thus, AG dependencies that *span across chips* (as reflected in the global order of same-address cachelines) must be preserved with an external protocol (across MCs), along the lines of the cross-chip LAD protocol [18].

D. Discussion: Recovery

A TSO guarantee for persistence has the potential to impact the recovery of applications that are based on lock-free algorithms and data structures. These applications do not necessarily fit the mould of epoch (or SFR) persistence. We leave this for the future work.

For other applications whose recovery is based on software-defined epochs, the software must indicate to TSOPER such epochs. This can be achieved with *marker* store instructions (uniquely identified by TSOPER) that are inserted in the store stream to control AG boundaries (e.g., start and freeze AGs). In the AGB, markers assemble collections of AGs into super groups that correspond to the software-defined epochs. TSOPER is limited by the size of the super group that can fit in the AGB, thus LAD-inspired solutions [18] can be adopted.

III. Atomic Group Ordering

A naïve approach to maintain TSO persistency when any local modification of an atomic group must be exposed, is to "stop the world" and persist the AG; we denote this STW. This means that both local and remote cores are prevented from making further modifications until the atomic group is buffered in the AGB for persist. In contrast to the naïve STW approach, TSOPER allows coherent operations to continue *after* an atomic group is frozen. This is shown in Fig. 6.

Atomic groups grow when stores exit the store buffer and write to the cache. This includes cachelines brought into the cache as a result of the write misses that are caused by the stores. Cache0 expands its AG by requesting `b` and `c` from cache1 and cache2 respectively. The act of accessing the cachelines in cache1 and cache2, (Fig. 6① and ②), freezes the respective

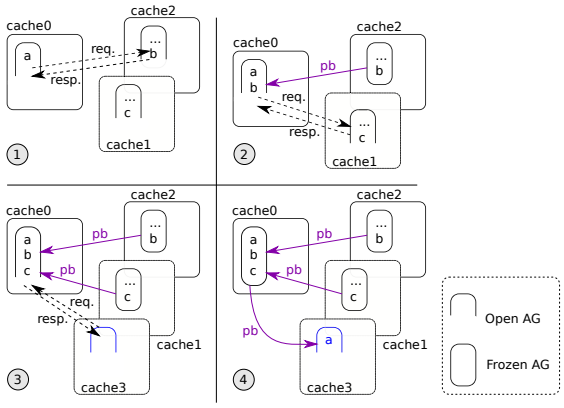


Fig. 6: Establishing pb dependencies among AGs.

AGs and creates *persists-before* (pb) *dependencies* among the atomic groups. Specifically, the AGs in cache1 and cache2 must persist before the AG of cache0:

$$AG_1 \xrightarrow{pb} AG_0 \text{ and } AG_2 \xrightarrow{pb} AG_0$$

Finally, cache3 requests a from cache0 (Fig. 6③) and freezes the AG of cache0 (Fig. 6④). This creates a pb between the AG of cache3 and that of cache0:

$$AG_0 \xrightarrow{pb} AG_3$$

The invariant here is: *An AG can create arbitrary* ***incoming pb dependencies*** *from other AGs by accessing remote cachelines, as long as it is not forced to reveal its modifications. On the first access to its own cachelines, i.e., on the first* ***outgoing pb dependence****, the AG is frozen and cannot create any new* incoming *dependencies.* Additional *outgoing* dependencies can be created even after an AG is frozen, when remote AGs access (as allowed by coherence) its cachelines.

A. The Role of the Reads

While it is obvious that an AG includes locally-modified cachelines, i.e., cachelines accessed with the intent to write, it is also the case that an AG includes cachelines that are accessed only for reading but belong to remote AGs. An example is shown in Fig. 7. We conservatively assume that any store following a load (e.g., st c that follows ld b in Fig. 7) may depend on the value accessed by the load. If the load brings to the cache a cacheline belonging to an external AG, and this cacheline is not included in the local AG (Fig. 7①), a pb dependence cannot be established between the AGs. Yet, concurrently persisting the AGs of Fig. 7① is an error as st c depends on the value written by st b in the second thread (on the right). This means that the currently expanding AG in the first cache (on the left) must include the cacheline that is read in order to encode the dependency to the remote AG. As we show in the next section, the dependencies of an AG are encoded in the cachelines it contains. Symmetrically to writes, a read freezes a remote atomic group.

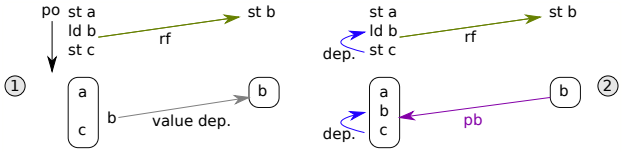


Fig. 7: Reads: ① Not including cacheline [b] in the AG on the left fails to establish a pb dependence to the AG on the right. ② Cacheline [b] included in the AG: a correct pb dependence is established.

B. Cache and Directory Evictions

Similarly to requests from other caches (cores), cache evictions or directory evictions have the potential to expose modifications within an AG to the outside world. Cache and directory evictions differ from requests of other caches, as they are not directly related to an expansion of another AG.

Potentially, a future AG that incorporates an evicted cacheline could establish a pb dependence to the AG that lost this cacheline. This is the approach taken by Joshi et al. [22], by keeping epoch information along with LLC cachelines. However, this is complicated as it requires important dependence information to be maintained in the face of LLC or directory evictions, which is not addressed in their work [22].

Instead, we opt for a different, more direct and simpler, solution with the help of some extra buffering: *We immediately freeze and persist an AG that suffers an eviction.* To hide the eviction latency —i.e., avoid delaying the replacement— we move the evicted cacheline into an *eviction buffer* on the side. The cacheline remains there until it persists, in accordance to the AG's pb dependencies. During that time, the cacheline still behaves as a member of the AG. Only after the AG is persisted, we perform the eviction at the coherence protocol level. A small eviction buffer typically suffices: Most items that are waiting to persist were recently written and, thus, less likely to be selected for eviction and cause pressure in the eviction buffer by occupying its entries for prolonged time.[3]

Similarly, an evicted directory entry is moved to a small buffer to immediately free up its place in the directory and expedite the replacement. It remains buffered until the affected cachelines have persisted. Potentially this can take a long time, but directory evictions are rarer and thus the buffering needed does not increase inordinately.

C. Deadlock Freedom

Unlike BSP [22], which uses a detection-and-avoidance policy to *break* deadlocks, in TSOPER, all deadlocks are *prevented* from happening by the inherent design of the system. BSP breaks and starts a new epoch and notes a dependency when it detects the use of a cacheline from an active epoch in another thread. This dependency is only resolved when the epoch of the other thread is eventually persisted. TSOPER instead pro-actively freezes an open AG in another thread if a cacheline is requested. Dependencies are thus resolved as soon as possible once detected.

[3]A 16-entry eviction buffer does not experience any pressure in our evaluations.

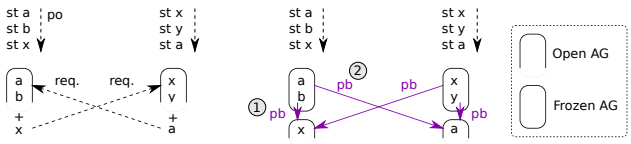


Fig. 8: Racing misses cannot create cycles among atomic groups because a cacheline joins an atomic group *after* the response arrives from the remote cache. By that time, the requests have frozen the old AGs and the newly installed cachelines start new AGs in the caches.

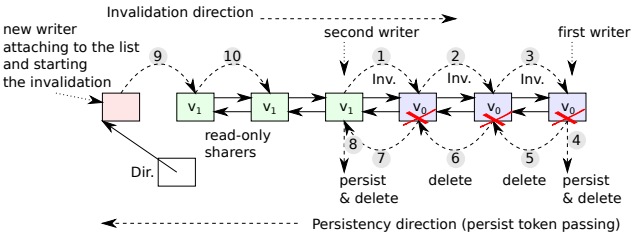


Fig. 9: Sharing-List Persistency.

In TSOPER, absence of *persists-before cycles* that would lead to *persist deadlocks* is established by two policies:

1. *Persist-before dependencies* are always formed along the flow of (logical) time. *Intra*-cache dependencies are formed between the last frozen AG and the newly opened AG (Fig. 8①). *Inter*-cache dependencies are formed along *responses* to coherence requests, i.e., a cacheline is added to an AG only once it has arrived, not at the time of the request (Fig. 8②).
2. All *incoming* pb dependencies of an AG are formed *before* all *outgoing* pb dependencies. This is because AGs can only create incoming pb dependencies as long as they are open, and they are frozen *before* they service the *first* request for one of their modified cachelines.

Thus, (logical) time passes with every established pb dependency and between the last incoming and first outgoing dependency of an AG. This naturally establishes a *cycle-free* dependence graph among all AGs.

IV. Dependence-Tracking Coherence

As we have shown, *persists-before* dependencies among AGs must be tracked to correctly persist in TSO. A key insight of our work is that *tracking* pb dependencies can be accomplished by the coherence protocol at the level of individual cachelines: The dependencies of an AG (to other AGs) *are* the dependencies of its cachelines. We next show that the desired properties for AG dependence tracking are encapsulated in sharing-list coherence protocols, such as SCI [20]. Two properties of a sharing list which are important in our case are that: (1) a sharing list naturally captures the serialization of operations that happens at the directory; and (2) it keeps this information entirely in the L1 caches. This sharing list "order" is used to enforce TSO persistency by ordering the persist of entire atomic groups. Our L1 coherence-based dependence tracking subsumes the dependence tracking mechanisms of Joshi et al. [22] that span the whole cache hierarchy (i.e., the IDT and LLC epoch information).

A. Sharing-List Persistency

One of the main contributions of our work is to show that we can use the concept of the sharing list to seamlessly provide *TSO of the persists for a single (cacheline-granularity) address.* Atomic groups, discussed next, order persists across multiple (cacheline-granularity) addresses. Different writers of a shared block queue up, one after the other, in the same sharing list, and perform their persists in the order in which they joined the sharing list. Contrary to the typical coherence protocols, we do not discard cachelines after they are invalidated; instead, we keep them until we persist their version of the data. Thus, sharing lists are not destroyed at invalidation; they are dismantled by the orderly persist of their locally-modified cachelines. Persistency is enforced belatedly, lagging behind coherence, but following the same order. Our approach is distilled in three principles:

1. Non-destructive invalidations Invalidations do not remove cachelines from the sharing lists. Instead, when invalidated, such cachelines are forced to persist (if they carry locally-modified data). They remain in the sharing list in an invalid state, until their local modifications are persisted (cf. Fig. 9). Being invalid means that their version of the data is not available to anyone (either local or remote core) to access. It is strictly there to persist in order.

2. Multiversioning Not removing invalidated cachelines, implies that a sharing list can contain multiple versions of the same data simultaneously. Only one version is *valid* (and *current*) and can be accessed: the version at the head of the sharing list. All other versions are invalid (and *stale*) but remain in the sharing list until they are persisted. In Fig. 9, at some point in time, two versions (v_0 and v_1) co-exist.

3. Tail-to-head persist Invalid cachelines in a sharing list can only persist when they become the *tail* in their sharing list. After persisting, invalidated tails disconnect from the sharing list, making the next cacheline (towards the head) the new tail. One way to visualize our approach is to consider that the tail of each sharing list owns a *persist token* that is passed up the list, *from tail to head*. Invalidated, locally-modified tails first persist and then pass the token; invalidated *unmodified* tails immediately pass the token and disappear. Cachelines are deleted once they become both invalid and pass on their persist token. Figure 9 shows an example.

Sharing-list persistency is reminiscent of the "*Queue on Lock Bit*" (*QOLB*) hardware queue lock, proposed by Goodman et al. [17] and adopted in SCI. Indeed, this is the case. The novel aspects of our work are:

- We allow multi-versioning to go on while building the sharing list, while QOLB's purpose is to spin (locally) on all cachelines on the list, except the tail which is the lock holder.
- In QOLB, all cachelines on the list *except the tail* are waiting to become valid, while, in our case, all the cachelines on the list *except the head* are invalid and waiting to be deleted.

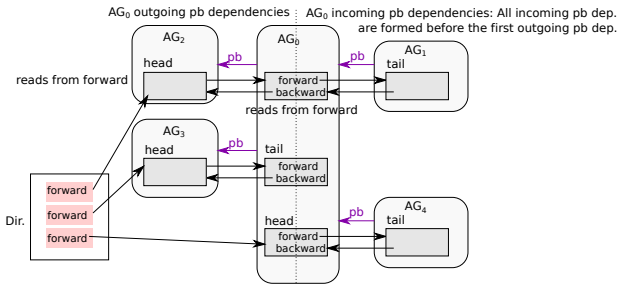


Fig. 10: The pb dependencies of AG_0 are encoded by the sharing lists of its three cachelines.

If desired, and convenient, protocol functionality can double up for both schemes, but we leave this to be examined in future work.

B. Putting it All Together

The last remaining piece of the puzzle is to bring together the concept of the atomic group persistency and the concept of the sharing-list persistency. Their combination gives us hardware TSO persistency. As we have alluded, the incoming and outgoing pb dependencies of an AG are, in fact, encoded in the cachelines it contains. They are the sharing list *forward* and *backward* pointers of its cachelines.

Figure 10 shows this relationship: A valid forward pointer is an *incoming* pb dependence: we have read local modifications from an AG towards the tail. A valid backward pointer is an *outgoing* pb dependence: an AG towards the head has read out local modifications.

Finally, to put everything together, we need to clarify how an AG persists based on the sharing list state of its cachelines. The two invariants we must keep for an AG are:

1. *All incoming* pb *dependencies must be satisfied* (all atomic groups on which the AG depends on must be persisted) *before any of the AG's cachelines are persisted.*
2. *No cacheline of any AG that depends on an AG can be persisted before all of that AG's cachelines are persisted.*

An AG freezes and starts the persist process when another AG requests one of its cachelines containing locally-modified data. The persist process consists of a phase in which the cache controller waits (if needed) for *all* cachelines of the AG to become tails and a phase in which the cache controller persists the cachelines of the AG to the AGB. We use a counter to keep track of the number of cachelines in an AG that are waiting to become tail throughout the AG's lifetime.[4] Recall that all AG cachelines eventually become tails because they trigger their predecessor AGs to freeze and persist. Once the AG is frozen and the "waiting to become tail" counter reaches 0, the AG is ready to persist. The persist token is passed immediately on persist: as soon as a cacheline is buffered in the AGB it leaves the sharing list, making the next-in-list cacheline the tail.

[4]In the evaluation we use eight counters per cache so the cost is negligible.

TABLE I: System configuration.

Private L1 I&D caches	32 KB, 8 ways, 4 hit cycles, 64-byte block size, 16-entry eviction buffer
Private L2 cache	256 KB, 8 ways, 12 hit cycles, 16-entry eviction buffer
Shared LLC (8 banks)	1 MB per bank, 8 ways, 35 hit cycles
Directory (8 banks)	512 sets, 8 ways (200% coverage)
Atomic Group Buffer	8 AGBs, each holds 160 cachelines (2×80-cacheline AGs)
NVM size	4.0 GB, 4096-byte page size
NVM ranks	8 DDR (8 MCs)
NVM write / read delay	360 / 240 cycles
NoC	Fully connected, 128-bit datapath, 5-flit data msg/1-flit ctrl msg, 6 cycles per hop

V. Evaluation

In this section, we evaluate how TSOPER performs against state-of-the-art approaches for strict and relaxed persistency.

System configuration: Our simulation infrastructure is driven by a Sniper [6] front-end that feeds instructions to an in-order processor model. Our sharing list coherence (SLC) protocol is implemented using the SLICC language [29]. We use the MOESI_CMP_directory protocol (found in gem5/GEMS distributions) as an example to indicate that SLC is *not* unreasonably complex compared to a widely used directory protocol. The SLICC implementation of SLC compared to the standard MOESI protocol requires fewer base states (15 vs. 25), fewer transient states (24 vs. 64), slightly higher SLICC actions (133 vs. 127), and far fewer SLICC transitions (148 vs. 264). We simulate a multi-core processor consisting of eight cores. The private caches have eviction buffers for cachelines pending to persist, sized to prevent a bottleneck. The processors are connected to Ruby, a cycle-accurate memory hierarchy model. The interconnection network is modelled with GARNET [1]. Details of the simulated system appear in Table I.

Memory and AGB: We model 8 NVM ranks, each as a double data-rate (DDR) rank with 360(240) cycles write(read) latency [22]. As depicted in Fig. 5, we simulate eight distributed AGBs, one per NVM rank, with one centralized AGB arbiter. We follow BSP [22] to implement the handshaking protocol between the centralized AGB arbiter and the distributed AGBs. The AGB arbiter tracks free space across AGBs and accepts AGB reservations from up to eight different L1 caches in parallel when enough space is present. Each AGB is simulated as a (power-backed) circular SRAM buffer, which is associated to the memory space slice of one PM rank.

Benchmarks: We run applications from PARSEC 3.0 [4] and Splash-3 [39] suites with both small (*barnes*, *cho.*, *fft*, *freq.*, *lu*, *stream.*, *swap.*, and *vips*) and large (*black.*, *body.*, *can.*, *dedup*, *ferret*, *fluid.*, *ocean*, *radio.*, *radix*, *raytrace*, *volrend*, *water*, and *x264*) inputs and present results for their region of interest.

Systems: We compare against the following PM systems:

1. Baseline A system with SCI-like [20] sharing list coherence (SLC) that builds sharing lists in the L2 (private) caches with no persistency support. In our simulation infrastructure, we confirm previous studies [14] that SLC carries a ∼3% overhead compared to MESI. However, in future work this performance

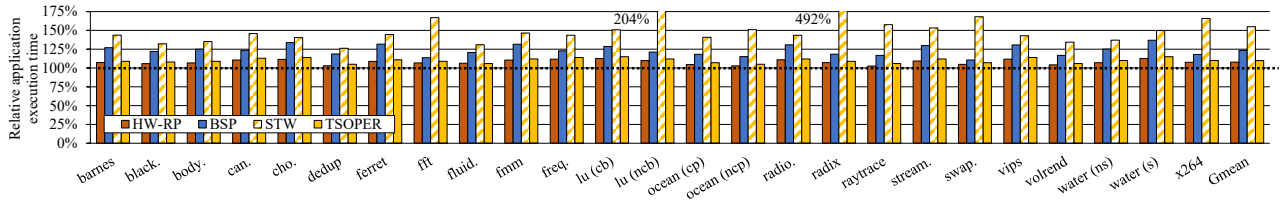


Fig. 11: Relative application execution time comparison across baseline, HW-RP, BSP, STW, and TSOPER.

gap can be addressed with a hybrid protocol that combines a MESI component with an SLC component, especially in cases where only a small subset of addresses are persisted, as for example in the WHISPER benchmark suite which persists only ∼4% of the stores [30]. The SLC component would then be used exclusively for such addresses. Since in our workloads, all addresses are persisted, we indiscriminately use SLC for all coherence.

2. HW-RP Because we propose a hardware approach, we cannot directly compare with a software implementation, such as SFR persistency [15], that: (1) includes significant instruction overhead (both in the static binary and during execution); (2) relies on fine-grain persist operations (which we do not encounter in actual hardware implementations); and (3) implements its own unconstrained redo logging that effectively doubles the writes to NVM. Instead, we opt to characterize the cost of TSO. We compare TSOPER to a hardware implementation of a hypothetical relaxed-persistency model, which we call HW-RP. HW-RP imposes no order among persist operations within synchronization-free regions. Similarly to SFR persistency, in HW-RP: (1) synchronization must be exposed to the hardware by the programmer, and (2) persist order is enforced across synchronization points. HW-RP persists at cacheline granularity. We disregard false-sharing that may creep up in various programs, giving HW-RP an advantage. Evictions of dirty lines are counted as spontaneous persists. As in TSOPER, persists are buffered in the AGB but the AGB size is *unconstrained* to fit arbitrary large SFRs. There is no other undo (or redo) logging implemented, meaning that there are no guarantees for atomic persistency of complete synchronization-free regions. The point of HW-RP is to remove the *ordering overhead* of TSOPER and thus allow its characterization.

3. BSP We implemented BSP following the paper of Joshi et al. [22]. Qualitative differences between BSP and TSOPER have been discussed in §I.

4. STW and TSOPER The "stop-the-world" version of TSO persistency and our full proposal as described in §III and §IV. In both STW and TSOPER, we set a hard limit of 80 cachelines per AG for the atomic groups.

A. Performance

Figure 11 reports application execution time for HW-RP, BSP, STW, and TSOPER, normalized to the baseline SLC protocol. As shown, by stalling both local cores and remote cores, STW manages the worst performance among all approaches, adding on average 53% overhead to application execution time. The two worst cases are *lu (ncb)* and *radix*, where STW increases total execution time by 104% and 392%. This is because the total NVM persist volume of *radix* and *lu (ncb)* is high and both programs create a large number of AGs, causing frequent NVM writes. In STW, this can lead to frequent stalls of both local and remote cores.

In contrast to STW, the other systems, HW-RP, BSP and TSOPER, maintain memory persistency in a non-blocking way, (i.e., both local and remote cores are allowed to continuously execute without stopping). However, due to L1 and LLC exclusion time shown in Fig. 1, BSP performs worse than both HW-RP and TSOPER.

Both HW-RP and TSOPER add less overhead to the application execution time than BSP. On average, HW-RP increases application execution time by about 7% (max 13%) and TSOPER by 10% (max 15%). In contrast, BSP increases execution time by 22% (max 34%). TSOPER achieves this result by further decoupling core execution from persistency, using the SLC protocol to reduce L1 exclusion time and the AGB to reduce LLC exclusion time.

B. Comparison to BSP

To understand the difference between BSP and TSOPER, we model two intermediate mechanisms, BSP+SLC and BSP+SLC+AGB, as stepping-stones from BSP to TSOPER.

BSP+SLC is implemented by replacing the coherence protocol of BSP with *our first innovation*, the SLC protocol. This enables L1 multi-versioning, reducing L1 exclusion time.

BSP+SLC+AGB further enhances BSP+SLC with *our second innovation*, the AGB arbiter and distributed AGBs, reducing LLC exclusion time. However, BSP relies on large 10,000-store AGs that are rarely broken (only when a possible possible deadlock is detected) that will not fit in a small AGB. Thus, BSP+SLC+AGB represents an idealized case (infeasible in current technology) that models an *unbounded* AGB, able to fit the very large AGs of BSP.

Figure 12 compares the execution time of BSP, BSP+SLC, BSP+SLC+AGB, and TSOPER, with results normalized to TSOPER.

We can observe that from BSP to BSP+SLC, reducing the L1 exclusion time achieves an average of 3% (max 7%) execution time improvement. Unlike BSP, BSP+SLC allows modification of any remote dirty L1 cacheline in the local L1 without waiting for the remote cacheline to be written out to LLC. We also observe that in applications

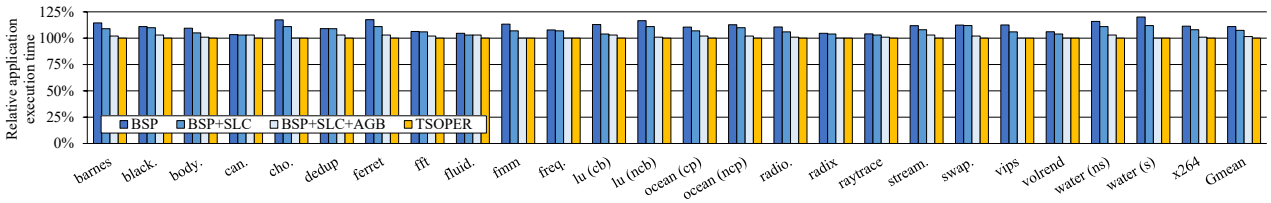


Fig. 12: Relative application execution time comparison across baseline, BSP, BSP+SLC, BSP+SLC+AGB, and TSOPER.

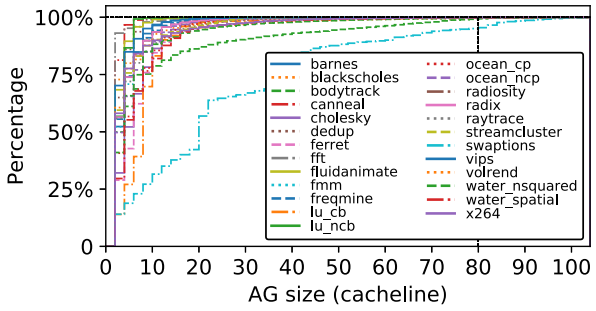


Fig. 13: AG size cumulative histogram.

with few simultaneous writes (such as *black.*, *swap.*), BSP and BSP+SLC achieve similar results. It is worth noting here that the L1 buffering achieved by SLC is reflected in the average length of the *persist lists* which reaches ~ 4 across all benchmarks, while the length of the corresponding *coherence sharing lists* is, on average, below ~ 2. Depending on benchmark characteristics, the average size of the persist lists varies, e.g., from ~ 2 in *dedup*, to ~ 4 in *x264* and to ~ 6 in *body*.

Further, by reducing LLC exclusion time, BSP+SLC+AGB enhances results of BSP+SLC on average by 7% (max 10%).

We can observe in Fig. 12 that BSP+SLC+AGB still suffers on average another 3% (max 5%) worse application performance than TSOPER. This is because in BSP+SLC+AGB, epochs are set to static sizes (10,000 stores [22]) unless deadlock happens. According to our study shown in Fig. 13, we find that throughout all benchmarks, AG size is seldom (less than 1%) larger than 80 cachelines. Compared to the AG size in TSOPER, large epoch size in BSP+SLC+AGB introduces higher serialization overhead when transmitted to the NVM, hurting performance. With epoch size changed to 80 cachelines, we find that BSP+SLC+AGB indeed achieves close results to TSOPER. This also reveals that by optimizing epoch size only (without using SLC nor AGB), the improvement potential for BSP is limited to between 3% and 5% of execution time.

C. Total Volume of Persists

Figure 14 shows the write traffic caused by coherence and persistence, respectively. The dotted bars represent normal downgrades and writebacks to LLC banks (coherence write), while the solid bars represent writes to AGBs and NVMs (persistence write). All results are normalized to the SLC baseline (not shown in Fig. 14). The baseline does not persist anything so it corresponds in its entirety to normal coherence downgrade and writeback traffic (100%). We can make the following observations:

1. BSP, STW and TSOPER do not cause any significant difference with respect to coherence and persistence traffic, i.e., persistence writebacks are about as many as coherence writebacks (but network traffic appears doubled because they are routed to different destinations). This is consistent with the illustration in Fig. 1, where one AG is both written back to LLC for coherence and to NVM for persistence, albeit at different times.
2. Compared to STW and TSOPER, HW-RP produces much higher persist traffic. Although HW-RP and TSOPER have the same level of persist re-ordering, HW-RP coalesces much less than TSOPER, as will be explained later.
3. By allowing coalescing as much as possible, BSP, STW, and TSOPER reduce the persist traffic significantly to the level of the coherence downgrade and writeback traffic.

D. Store Coalescing and SFR/AG size

Figure 15 compares the SFR size in HW-RP and AG size in TSOPER using *ocean_cp* (the worst case application in Fig. 14). *Ocean_cp* simulates large-scale ocean movements where waveforms are partitioned into grids. Grids are periodically synchronized, thus the SFR and AG size exhibits periodical increases (during non-synchronization regions) and decreases (during synchronization regions), as can be observed in the timeline figures. The cumulative histograms show that HW-RP creates a large number (over 90%) of small sized SFRs (=1 store) and a small number (less than 3%) of large sized ones (over 2.5K stores). This is because *ocean_cp* is synchronized using critical sections (CS). In HW-RP, a CS itself and the intermediate zone between two CSs are both counted as SFRs. On one hand, CS-oriented SFRs are limited in size, with one CS usually issuing one store updating one shared variable. On the other hand, SFRs created by the intermediate regions are free in size. While large SFRs enjoy high coalescing rate, small SFRs lose the benefits of coalescing due to limited stores. In TSOPER, AGs are dynamically created by data sharing among cores. Because cachelines stay in L1s as long as possible until eviction or until they are exposed, TSOPER achieves a high coalescing rate, and a lower NVM write volume than HW-RP.

VI. Related Work

Byte-addressable NVM technologies, such as PCM [28], [34], STT-RAM [24], [27], and ReRAM [2], have enabled persistent

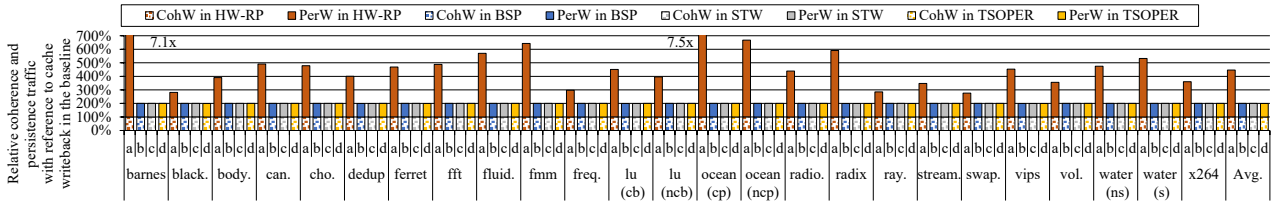


Fig. 14: Coherence write (CohW) and persist write (PerW) traffic. Case a denotes HW-RP, b BSP, c STW, and d TSOPER.

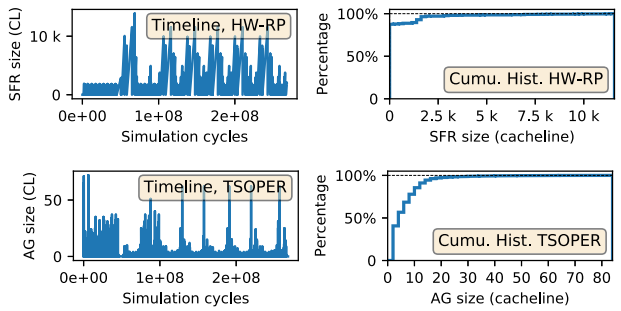


Fig. 15: SFR, AG size timeline and cumulative histogram through *ocean_cp* execution.

memory (*PM*) to approach the performance and capacity of DRAM [3], [8], [31], [36], [38], [40], [44]. Current research in this area mainly focuses on defining persistency models (i.e., models which define *persist ordering* and *atomicity*). Related work on persistency can be categorized into: (1) data structure level persistency (e.g., [9], [12], [43], [46]), (2) ISA level persistency, and (3) language level persistency. Further, because logging has been widely used in PM to ensure persistency atomicity, various works propose efficient logging.

ISA Level Persistency: ISA level persistency depends on special instructions or transactional memory to write persists into PM and keep them in order. Recently, Intel has announced a new persistency model and a corresponding Development Kit (PMDK). The ordering effects of Intel-x86 instructions (`flush`, $\texttt{flush}_{opt}$, `CLWB`) on persists were recently described by Raad et al. [33]. Ren et al. propose a checkpointing mechanism named ThyNVM [35] to hide the log writing time to PM. To achieve this, ThyNVM dynamically determines checkpoint granularity so that the trade-off between application stall time and metadata storage overhead can be found. Volos et al. [44] proposed a lightweight mechanism implementing programming interfaces for PM that provides logging-based consistent update to persistent data structures, and a durable memory transaction mechanism that enables consistent updates of arbitrary data structures.

Language Level Persistency: Such approaches argue for extending the language-level memory model to provide guarantees on the order of persistent writes. Atlas [8] provides persistency semantics for lock-based multi-threaded C++ programs. It guarantees failure atomicity for the *coarse-grained* outermost critical section (CS). Compared to our proposal, ATLAS does not consider PM support outside critical sections while we continuously persist in TSO. In contrast to ATLAS, acquire-release persistency (ARP) [25], [26] maintains persistency at *fine-grained* individual stores. Compared to ARP, our proposal provides strict TSO both within and outside epochs. ARP reorders persists within epochs (the region between lock acquire-release or release-acquire), but forbids persist reordering across epochs. Seeking the middle-ground between ATLAS and ARP, synchronization-free region (SFR) persistency [15] advocates for PM persistency at the granularity of the region between two synchronization operations. To ensure PM atomicity, SFR uses an undo-log based mechanism (coupled or decoupled to the SFR frontier) to roll back to the last durable SFR block in the PM. We have compared to SFR in previous sections.

Efficient Logging: Cohen et al. [10] and Joshi et al. [23] propose low-overhead logging cache coherence protocol and controller. Instead of manipulating cache controller, Doshi et al. [13] implement dedicated hardware controller off-loading PM logging to run in parallel with application execution. Jeong et al. [21] propose redo-based logging (ReDU), which performs direct and asynchronous in-place data update to NVM. ReDU exploits a small region of DRAM as a write-cache to remove NVM writes from the critical path. Shin et al. [42] observe that PM instructions (such as `flush`, `CLWB`, etc.) in conjunction with `SFENCE` incur long pipeline stalls. They reduce such stalls by proposing a speculative execution mechanism: rather than letting the `SFENCE` stall the core to wait for persists to become durable, the mechanism checkpoints the architectural state and retires the `SFENCE` speculatively so that the core does not need to stall. In our work, we rely on a modified feature of Intel architectures, the WPQ, or AGB in our case, to provide logging functionality.

VII. Concluding Remarks

We describe a new approach to strict persistency: after a crash, we offer an observable total order of the stores executed before the crash, using *limited* (potentially little) hardware buffering in the form of a TSO persist buffer. We achieve this by bundling persists (at cacheline granularity) in atomic groups (AGs), and enforcing an order among the AGs that respects TSO consistency. We show that enforcing order among AGs, as opposed to relaxing all order of cacheline persists between synchronizations, has a negligible impact on performance. We further propose an elegant way to detect and encode the ordering dependencies of AGs by using *sharing-list coherence*. In this respect, we do not add cost to track the order of AGs.

Finally, we show that we achieve a greater decoupling of coherence and persistence than BSP, the prior state-of-the-art solution for strict persistency. TSOPER offers continuous TSO persistency, independently of synchronization. One can build on top of it (with other forms of logging) to offer more stringent language-level guarantees, for example, consistent persists across synchronization fronts.

REFERENCES

[1] N. Agarwal, T. Krishna, L. Peh, and N. K. Jha, "GARNET: a detailed on-chip network model inside a full-system simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS 2009. IEEE Computer Society, Apr. 2009, pp. 33–42. [Online]. Available: https://doi.org/10.1109/ISPASS.2009.4919636

[2] H. Akinaga and H. Shima, "Resistive random access memory (ReRAM) based on metal oxides," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, 2010. [Online]. Available: https://doi.org/10.1109/JPROC.2010.2070830

[3] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *ACM International Conference on Management of Data*, ser. SIGMOD 2015, 2015, pp. 707–722. [Online]. Available: https://doi.org/10.1145/2723372.2749441

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT 2008. ACM, Oct. 2008, pp. 72–81. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[5] H. Boehm and S. V. Adve, "Foundations of the C++ concurrency memory model," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, ser. PLDI 2008. ACM, Jun. 2008, pp. 68–78. [Online]. Available: https://doi.org/10.1145/1375581.1375591

[6] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC 2011. ACM, Nov. 2011, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2063384.2063454

[7] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "BulkSC: Bulk enforcement of sequential consistency," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA 2007. ACM, Jun. 2007, pp. 278–289. [Online]. Available: https://doi.org/10.1145/1250662.1250697

[8] D. R. Chakrabarti, H. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA 2014. ACM, Oct. 2014, pp. 433–452. [Online]. Available: https://doi.org/10.1145/2660193.2660224

[9] N. Cohen, D. T. Aksun, and J. R. Larus, "Object-oriented recovery for non-volatile memory," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 153:1–153:21, 2018. [Online]. Available: https://doi.org/10.1145/3276523

[10] N. Cohen, M. Friedman, and J. R. Larus, "Efficient logging in non-volatile memory by exploiting coherency protocols," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 67:1–67:24, 2017. [Online]. Available: https://doi.org/10.1145/3133891

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, ser. SOSP 2009. ACM, Oct. 2009, pp. 133–146. [Online]. Available: https://doi.org/10.1145/1629575.1629589

[12] T. David, A. Dragojević, R. Guerraoui, and I. Zablotchi, "Log-free concurrent data structures," in *USENIX Annual Technical Conference*, ser. USENIX ATC 2018, Jul. 2018, pp. 373–385. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/david

[13] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for SCM with a non-intrusive backend controller," in *IEEE International Symposium on High-Performance Computer Architecture*, ser. HPCA 2016. IEEE Computer Society, 2016, pp. 77–89. [Online]. Available: https://doi.org/10.1109/HPCA.2016.7446055

[14] R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Are distributed sharing codes a solution to the scalability problem of coherence directories in manycores? An evaluation study," *Journal of Supercomputing (JSC)*, vol. 72, no. 2, pp. 612–638, Feb. 2016. [Online]. Available: https://doi.org/10.1007/s11227-015-1596-4

[15] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. ACM, Jun. 2018, pp. 46–61. [Online]. Available: https://doi.org/10.1145/3192366.3192367

[16] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *Proceedings of the 47th Annual International Symposium on Computer Architecture*, ser. ISCA 2020. ACM, Jun. 2020, pp. 652–665. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00060

[17] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficient synchronization primitives for large-scale cache-coherent multiprocessors," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 1989. ACM Press, Apr. 1989, pp. 64–75. [Online]. Available: https://doi.org/10.1145/70082.68188

[18] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 2019. ACM, Oct. 2019, pp. 466–478. [Online]. Available: https://doi.org/10.1145/3352460.3358321

[19] "Intel® 64 and IA-32 architectures software developer's manual (combined volumes)," Oct. 2019, Order Number: 325462-071US. [Online]. Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[20] D. V. James, A. T. Laundrie, S. Gjessing, and G. S. Sohi, "Distributed-directory scheme: Scalable coherent interface," *IEEE Computer*, vol. 23, no. 6, pp. 74–77, Jun. 1990. [Online]. Available: https://doi.org/10.1109/2.55503

[21] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 2018. IEEE Computer Society, Oct. 2018, pp. 520–532. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00049

[22] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 2015. ACM, Dec. 2015, pp. 660–671. [Online]. Available: https://doi.org/10.1145/2830772.2830805

[23] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: atomic durability in non-volatile memory through hardware logging," in *IEEE International Symposium on High-Performance Computer Architecture*, ser. HPCA 2017. IEEE Computer Society, Feb. 2017, pp. 361–372. [Online]. Available: https://doi.org/10.1109/HPCA.2017.50

[24] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. M. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno, "2Mb spin-transfer torque RAM (SPRAM) with bit-by-bit bidirectional current write and parallelizing-direction current read," in *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, ser. ISSCC 2007. IEEE, Feb. 2007, pp. 480–617. [Online]. Available: https://doi.org/10.1109/ISSCC.2007.373503

[25] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA 2017. ACM, Jun. 2017, pp. 481–493. [Online]. Available: https://doi.org/10.1145/3079856.3080229

[26] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, W. Wang, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language support for memory persistency," *IEEE Micro*, vol. 39, no. 3, pp. 94–102, 2019. [Online]. Available: https://doi.org/10.1109/MM.2019.2910821

[27] E. Kultursay, M. T. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS 2013. IEEE Computer Society, Apr. 2013, pp. 256–267. [Online]. Available: https://doi.org/10.1109/ISPASS.2013.6557176

[28] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory,"

IEEE Micro, vol. 30, no. 1, pp. 131–141, 2010. [Online]. Available: https://doi.org/10.1109/MM.2010.24

[29] M. M. K. 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," *SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005. [Online]. Available: https://doi.org/10.1145/1105734.1105747

[30] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with WHISPER," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2017. New York, NY, USA: ACM, 2017, pp. 135—-148. [Online]. Available: https://doi.org/10.1145/3037697.3037730

[31] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *ACM/IEEE 41st International Symposium on Computer Architecture*, ser. ISCA 2014. IEEE Computer Society, Jun. 2014, pp. 265–276. [Online]. Available: https://doi.org/10.1109/ISCA.2014.6853222

[32] A. Raad and V. Vafeiadis, "Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 137:1–137:27, 2018. [Online]. Available: https://doi.org/10.1145/3276507

[33] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, "Persistency semantics of the Intel-x86 architecture," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 11:1–11:31, 2020. [Online]. Available: https://doi.org/10.1145/3371079

[34] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4–5, pp. 465–479, 2008. [Online]. Available: https://doi.org/10.1147/rd.524.0465

[35] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO 2015, Dec. 2015, pp. 672–685. [Online]. Available: https://doi.org/10.1145/2830772.2830802

[36] A. Rudoff, "Programming models for emerging non-volatile memory technologies," *;login:*, vol. 38, no. 3, pp. 40–45, Jun. 2013. [Online]. Available: https://www.usenix.org/publications/login/june-2013-volume-38-number-3/programming-models-emerging-non-volatile-memory

[37] ——, "Deprecating the PCOMMIT instruction," Sep. 2016. [Online]. Available: https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction

[38] ——, "Persistent memory programming," *;login:*, vol. 42, no. 2, pp. 34–40, 2017. [Online]. Available: https://www.usenix.org/publications/login/summer2017/rudoff

[39] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS 2016. IEEE Computer Society, Apr. 2016, pp. 101–111. [Online]. Available: https://doi.org/10.1109/ISPASS.2016.7482078

[40] S. Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*, 1st ed. Apress Open, Jan. 2020.

[41] 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," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010. [Online]. Available: https://doi.org/10.1145/1785414.1785443

[42] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA 2017, Jun. 2017, pp. 175–186. [Online]. Available: https://doi.org/10.1145/3079856.3080240

[43] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, ser. FAST 2011. USENIX, Feb. 2011, pp. 61–75. [Online]. Available: http://www.usenix.org/events/fast11/tech/techAbstracts.html#Venkataraman

[44] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2011, Mar. 2011, pp. 91–104. [Online]. Available: https://doi.org/10.1145/1950365.1950379

[45] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 2020. IEEE Computer Society, Oct. 2020, pp. 496–508.

[46] J. Yang, Q. Wei, C. Wang, C. Chen, K. L. Yong, and B. He, "NV-Tree: A consistent and workload-adaptive tree structure for non-volatile memory," *IEEE Trans. Computers*, vol. 65, no. 7, pp. 2169–2183, 2016. [Online]. Available: https://doi.org/10.1109/TC.2015.2479621