BBB: Simplifying Persistent Programming using Battery-Backed Buffers

Mohammad Alshboul[1], Prakash Ramrakhyani[2], William Wang[2], James Tuck[1], and Yan Solihin[3]

[1]ECE, North Carolina State University: {maalshbo, jtuck}@ncsu.edu
[2]Arm Research: {prakash.ramrakhyani, william.wang}@arm.com
[3]Computer Science, University of Central Florida: yan.solihin@ucf.edu

***Abstract*—Non-volatile memory (NVM) is poised to augment or replace DRAM as main memory. With the right abstraction and support, non-volatile main memory (NVMM) can provide an alternative to the storage system to host long-lasting persistent data. However, keeping persistent data in memory requires programs to be written such that data is crash consistent (i.e. it can be recovered after failure). Critical to supporting crash recovery is the guarantee of ordering of when stores become durable with respect to program order. Strict persistency, which requires persist order to coincide with program order of stores, is simple and intuitive but generally thought to be too slow. More relaxed persistency models are available but demand higher programming complexity, e.g. they require the programmer to insert persist barriers correctly in their program.**

We identify the source of strict persistency inefficiency as the *gap* between the point of visibility (PoV) which is the cache, and the point of persistency (PoP) which is the memory. In this paper, we propose a new approach to close the PoV/PoP gap which we refer to as Battery-Backed Buffer (BBB). The key idea of BBB is to provide a battery-backed persist buffer (bbPB) in each core next to the L1 data cache (L1D). A store value is allocated in the bbPB as it is written to cache, becoming part of the persistence domain. If a crash occurs, battery ensures bbPB can be fully drained to NVMM. BBB simplifies persistent programming as the programmer does not need to insert persist barriers or flushes. Furthermore, our BBB design achieves nearly identical results to eADR in terms of performance and number of NVMM writes, while requiring two orders of magnitude smaller energy and time to drain.

I. Introduction

Non-volatile main memory (NVMM) is poised to augment or replace DRAM as main memory. Due to its non-volatility, byte addressability, and being much faster in speed than SSD and HDD, NVM can host persistent data in main memory [4], [10], [11], [39], [46], [51], [52], [74], [95], [96].

In order to utilize this non-volatility feature, it is critical to guarantee ordering of *persists*, i.e. the ordering of stores reaching persistent memory to become durable. This is specified through a persistency model [5], [23], [38], [43], [68]. Without an explicit guarantee, the persist order will follow the cache replacement policy instead of the program order in updating the persistent memory state, and this may lead to inexplicable results. Programmers rely on the persistency model to write both normal-operation code and post-crash recovery code, and to reason about how such code can keep persistent data in a consistent state [23], [27], [43], [54], [68], [88], [89]. In designing persistency models, it is generally accepted that there is a tradeoff between performance and programmability. For example, strict persistency requires persist order to coincide with program order of stores, while epoch persistency orders persists across epochs but not within an epoch. While a more relaxed persistency model can offer higher performance, adopting a more relaxed persistency model burdens the programmer with additional tasks, e.g. defining epochs.

Another persistency programmability challenge is caused by the gap between the point of visibility (PoV) and the point of persistency (PoP), which affects parallel programs (Figure 1). A store may become visible to other threads when the value is written to the cache, but does not persist until reaching the memory controller (MC)[1]. Furthermore, before persistency is ensured, a store value may be observed by another thread which may persist another value, resulting a non-consistent persistent memory state.

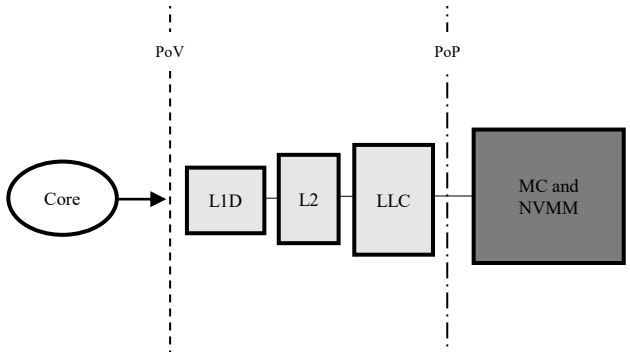

Fig. 1: Illustration for the gap between the Point of Visibilty (PoV) at the L1D cache, and the Point of Persistency (PoP) at the NVMM or MC.

Managing persist order and the PoV/PoP gap currently incurs substantial performance penalty and requires oner-

This research is supported in part through the following grants: Solihin was supported by NSF grant 1900724 and UCF. Alshboul and Tuck were supported in part by NSF grant CNS-1717486 and by NC State. Alshboul's PhD primary and co-advisor are Solihin and Tuck, respectively. Wang received funding from the European Union's Horizon 2020 research and innovation programme under project Sage 2, grant agreement 800999.

[1]We assume a base system with ADR [37], where an update becomes persistent when it reaches the write pending queue (WPQ) of the memory controller (MC).

ous effort from the programmer. For example, consider Intel PMEM [73], which provides programmers flush (clwb/-clflushopt) and fence (sfence) instructions. To guarantee persist ordering of two stores, clflushopt followed by sfence need to be inserted between them, delaying the issue of the second store (for a long time) until the first store reaches the MC. A missing flush or fence may cause bugs that are difficult to identify or reproduce due to their intermittency. Furthermore, it is not easy to debug a persistent program as a crash must be induced at different points of the program to check its persistent state correctness.

Thus, in this work, we ask the question: *Can the gap between PoV and PoP be closed inexpensively?* Closing this gap simplifies persistent programming and improves performance [90], because any committed store value persists and becomes visible simultaneously. One approach described in prior literature is to allow the PoV/PoP gap to exist, but hides the side effect if the gap may be exposed. In Bulk Strict Persistency (BSP) [43], if a store value has not persisted but is requested by another thread/core, it (and older stores) are persisted first before responding to the request. This complicates cache coherence and delays responses to external requests. Another approach being considered by industry is to close the PoV/PoP gap by using non-volatile caches (NVCache), where the entire cache hierarchy is added to the persistence domain. NVM technologies (e.g. STT-RAM, ReRAM, and PCM) [28], [48], [71] could be considered but they suffer from limited write endurance, high access latency, high write energy, and low write bandwidth [42], [61], [87]. These problems make NVCache more suitable for last level (or near last level) cache. However, the reduced PoV/PoP gap may improve performance but does not simplify programming. An alternative would be to use battery-backed SRAM caches through eADR [80]. However, eADR is expected to be costly as it requires a large battery to back the entire cache hierarchy [16].

In this paper, we propose a new approach to close the PoV/PoP gap. We call this the Battery-Backed Buffer (BBB). The key idea of BBB is to provide a persist buffer in each core next to the L1 data cache (L1D) that is non-volatile (backed by battery). A store is allocated an entry in the battery-backed persist buffers (bbPB) as it is written to cache, hence it becomes visible and persistent simultaneously. The stores in bbPB are then lazily drained from the bbPB to memory. If a crash occurs, the battery ensures bbPB can be fully drained to NVMM. The bbPB is sized to balance performance and battery cost. We find that a small number of entries (e.g. 32) is sufficient. BBB provides strict persistency semantics without the performance penalties associated with it. We explore a major design choice of whether to view bbPB as processor-side or memory-side structures, and discuss their tradeoffs. *Due to the complete elimination of PoV/PoP gap, persistent programming is simplified as flushes and fences are no longer necessary.* We show that with careful design, BBB's performance is nearly identical to eADR but with much lower hardware overhead because the bbPB size is much smaller than caches.

TABLE I: Comparison between several schemes for providing strict memory persistency ordering: Intel PMEM, Bulk Strict Persistency (BSP), eADR, and Battery-Backed Buffers (BBB).

Aspect	PMEM	BSP	eADR	**BBB**
SW Complexity	High	Low	Low	**Low**
Persist Inst.	clwb & fence	None	None	**None**
HW Complexity	Low	High	Low	**Low**
Strict pers. penalty	High	Medium	None	**Low**
Battery Needed	None	None	Large	**Small**
PoP location	WPQ/mem	Mem	L1D	**bbPB/L1D**

Table I contrasts different approaches for implementing strict persistency. PMEM's programming complexity is the highest due to the need to correctly and completely insert flushes and fences. BSP requires highly complex architectural support to give the illusion of strict persistency even though the underlying hardware allows non-ordered persists, and still incurs substantial performance penalty for strict persistency. eADR requires a large battery to support draining of the entire cache hierarchy. PMEM and BSP do not close the PoP/PoV gap as the memory or MC still serves as PoP. In contrast, BBB incurs low programming complexity (no flushes/fences are needed), low hardware complexity, negligible performance penalty for strict persistency, and requires only small battery to drain a few bbPB entries on a crash. BBB also closes PoP/PoV completely as bbPB is added next to the L1D cache.

Overall, the contributions of this paper are the following:

- A novel approach called Battery-Backed Buffer (BBB) to close the PoV/PoP gap, including bbPB design choices and coherence mechanisms.
- Estimation of energy and area needed for both BBB and eADR, showing BBB with two orders of magnitude improvement over eADR with regards to the energy and time costs for draining.
- Evaluation of BBB's effectiveness with different bbPB sizes.

The rest of the paper is organized as follows: Section II provides a relevant background. Section III introduces BBB and its design. The evaluation methodology is shown in Section IV, while the experiments results are reported in Section V. Section VI discusses relevant prior works. The paper concludes in Section VII.

II. Background

A. The Difficulty of Persistent Programming

In this section, we highlight how challenging it is for the programmer to write code that guarantees crash recoverability. This has already been discussed in several prior works that have emphasized how writing NVMM-friendly code can be very tedious and error-prone [1], [18], [56], [57], [64], [97].

As mentioned in Section I, the volatility of the cache hierarchy requires using persistency models to guarantee correct persistency ordering at the NVMM. These models usually require the programmer to add special instructions to the code to guarantee crash recoverability. These instructions often have two types of functionalities: (1) Sending updates to the NVMM,

```
1  void AppendNode(int new_val){
2  //create and initialize new node
3  node_t* new_node = new node_t(new_val);
4  //update new_node's next pointer
5  new_node -> next = head;
6  //update the linkedList's head pointer
7  head = new_node;
8  }
```

Fig. 2: Example code to add a node to the beginning of a linked list.

```
1  void AppendNode(int new_val){
2  //create and initialize new node
3  node_t* new_node = new node_t(new_val);
4  //update new_node's next pointer
5  new_node -> next = head;
6  //NEW: Persist new_node
7  writeBack(new_node);
8  persistBarrier;
9  //update the linkedList's head pointer
10 head = new_node;
11 //NEW: Persist head pointer
12 writeBack(head);
13 persistBarrier;
14 }
```

Fig. 3: Updated code to add a node to the beginning of a linked List.

which is done by flushing or writing back the corresponding cache block from the caches to NVMM (*writeBack*). Some examples of such instructions are (*clwb*) and (*clflushopt*) from x86 ISA, and (*DCCVAP*) and (*DCCVADP*) from Arm ISA. (2) After sending blocks to the NVMM, the second instruction type needed to guarantee persistency ordering is a barrier/fence instruction that ensures the subsequent instructions wait until the flushing is completed (*persistBarrier*). Examples of such barrier instructions are (*sfence*) and (*mfence*) for x86 ISA, and (*DSB*) and (*DMB*) for Arm ISA.

The main task of a programmer is to decide when and where to add these two types of special instructions. Figure 2 shows an example code to add a node to the head of a linked list. The code creates and initializes a new node (line 3), makes it to point to the current head node (line 5), and updates the head pointer to the new node (line 7). The code example is correct and works well if crash recoverability is not a concern. However, with NVMM, the code risks losing the entire linked list if stores persist out of the program order. For instance, the update to the head pointer may get persisted before the new node itself is persisted. If a crash occurs between the two persists, the new node will be lost (since it is still in the volatile caches), while the head pointer will still point to new node, which becomes invalid after the crash.

To make this code NVMM-friendly, the programmer may impose persist ordering by modifying the code as shown in Figure 3. Mainly, special instructions are added after storing the new node (line 7-8), and after storing the head pointer (line 12-13). With that, it is now guaranteed that the update to the head pointer will never be persisted until the update to the new node itself is persisted.

With BBB, this persist ordering problem will no longer exist, and the code shown in Figure 2 can still be safely used without any risk of persist ordering issues. This is because the new_node initialization (line 3) will become persistent immediately after its store is committed. Since the two stores are guaranteed to commit in program order, the store updating the head pointer (line 7) will commit after that, and hence the two updates are also going to automatically and instantly persist in that order. Note that the discussion about this code focuses on the persist ordering problem. Other programming problems (e.g. transaction semantics, permanent leaks) are out of the scope of this paper.

B. Non-Volatile Caches and eADR

As mentioned earlier, non-volatile caches (NVCaches) have been proposed and evaluated in literature [40], [58], [69], [77]. NVCaches rely on various NVM technologies, such as PCM, STT-RAM, and ReRAM [4], [28], [48], [52], [71], which differ in their access latency and density. However, they suffer from similar challenges as in NVMM, including limited write endurance. These problems will be more pronounced than NVMM because caches will be written at a much higher rate than memory, and the closer it is to the core, the higher the rate. Spin-Transfer Torque Random Access Memory (STT-RAM) has a relatively high write endurance level of 4×10^{12} writes, higher than alternatives such as Phase Change Memory (PCM) with 10^8 writes [52], [61], [71], [87], and Resistive Random Access Memory (ReRAM) with 10^{11} writes [4], [48], [61], [87]. However, their write endurance is still orders of magnitude lower than SRAM memory cells (about 10^{15}) [61], [71], [87]. Furthermore, they also suffer from high write energy, and higher access latency than SRAM caches. Finally, unless they are used for the entire cache hierarchy, they will still have PoV/PoP gap that complicates persistency programming.

Nevertheless, SRAM-based caches can become non-volatile by providing an additional energy source to create battery-backed caches. This energy source should be enough to implement *flush-on-fail* policy, where the entire cache hierarchy is drained to memory when a crash happens, and thus making the SRAM-based caches appear non-volatile. This approach enhances the ADR [37] guarantee from only covering the memory controller, to covering the entire cache hierarchy. Hence, it is named enhanced-ADR (eADR) [16], [77], [78]. Compared to NVCaches, eADR does not affect access latency, write energy, and write endurance. However, flush-on-fail requires a substantial amount of energy and time, resulting in considerable space and cost for the energy source (e.g. battery), and delaying crash recovery until after the draining is complete.

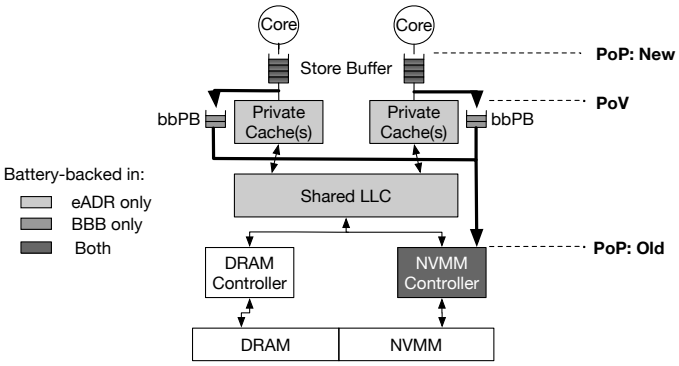

Fig. 4: BBB system overview.

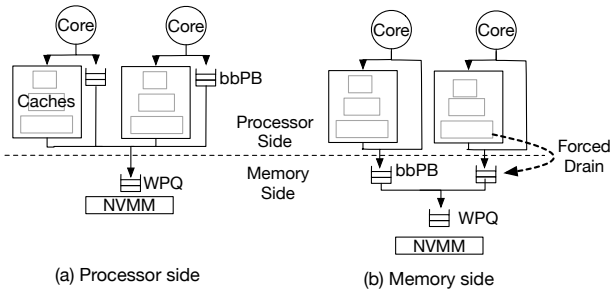

Fig. 5: bbPB logical view as a processor-side structure (a) vs. memory-side structure (b).

III. System Design

In this section, we discuss our proposed approach Battery-Backed Buffers (BBB), its mechanism, the design space, and the trade-offs associated.

A. High-Level Overview

A crucial component that BBB adds is the *battery-backed persist buffers* (bbPB), shown in Figure 4. The figure shows a multicore processor with flat physical address space that is divided into DRAM and NVMM. A portion of the NVMM address range is allocated for persistent data. Battery-backed components in eADR only are shown in light grey (all caches), battery-backed components in BBB only are shown in medium grey (bbPB), while battery-backed components in both eADR and BBB are shown in dark grey (store buffers and the write pending queue (WPQ) in the NVMM controller). Dataflow path for persisting stores follow the thick lines.

bbPB is located alongside the L1 data cache (L1D) of each core. Its first role is to serve as the Point of Persistence (PoP); any store allocated in the bbPB can be considered durable as bbPB ensures that the store will eventually reach NVMM through flush-on-fail draining; traditional persist buffers [50], [62] are volatile as they lose content if power is lost. With BBB, a battery provides sufficient energy to drain bbPB to the NVMM in the event of a crash.

The bbPB also plays a role in matching the Point of Visibility (PoV) and PoP, by moving PoP up from the MC to L1D. To achieve that, a store is allocated in bbPB at the same time the store goes to the L1D, after any cache miss or coherence state upgrade has been satisfied. Note that in some cases, the PoP needs to move further up to the store buffers (Section III-C). By closing the PoV and PoP gap, persistency programming is simplified as strict persistency can be achieved without explicit flushes and fences, and without the performance penalty of long-latency persist; a store instantly persists when allocated a bbPB entry. In contrast, traditional persist buffers (used with Buffered Epoch Persistency or BEP) [50], [62] have this gap and hence still require explicit persistency instructions. Stalls may still occur at epoch boundaries in BEP due to needing to wait the completion of persist buffer draining to NVMM.

BBB also requires much smaller battery than eADR which requires the entire cache hierarchy to be battery backed. The size of the battery depends on the worst number of entries that need to be drained at the time of crash, which is determined by the bbPB size (i.e. number of bbPB entries). Hence, the choice of bbPB size is an important parameter: it should be large enough to avoid stalling the core if bbPB is full, but should be small enough to keep the battery small and cheap.

To keep the battery small, bbPB is only used for stores that need to persist (i.e. *persisting store*). Non-persisting stores consist of stores that go to volatile memory (DRAM), or ones that go to NVMM but do not deal with persistent data. They can go directly to the cache hierarchy without involving bbPB. To distinguish store types, some prior work requires special instructions (e.g. NVload, NVStore) [58] to be used. Instead, we assume that regular store instructions are used. Persisting stores are distinguished from the pages that they access. We assume persistent data is allocated only in the heap using persistent memory allocation (e.g. *palloc*), which allocates such data in pages that map to page frames within the persistent portion of the physical address space. Persistent pages are allocated physically in the NVMM, while non-persistent pages can be allocated in either DRAM or NVMM.

B. Processor vs. Memory Side

So far we have not discussed how bbPB should interact with the rest of the memory hierarchy. One obvious choice is for the bbPB to be logically viewed as a *processor-side* structure, as it keeps track of persisting stores that need to drain to memory. This view is similar to traditional persist buffers [50], [62], but with added PoP guarantee. Figure 5(a) illustrates this view. On the other hand, bbPB can also be viewed as a structure on the *memory side*. The latter view makes sense as well because in the current architecture, only memory side structures such as the NVMM and write pending queue (WPQ) in the memory controller are in the persistence domain. As bbPB is added to the persistence domain, it can be thought of as a persistence domain extension of the write pending queues (WPQs) that are distributed in different cores.

The choice of the procesor- vs. memory-side organizations carry important consequences. The close physical proximity to the core makes the processor-side organization a more intuitive

choice. In such an organization, each bbPB entry corresponds to an (address, value) pair for each store instruction that needs to persist. Store granularity could be used (e.g. byte, word, doubleword). The stores need to be ordered in the bbPB because they have not yet reached the persistence domain. Coalescing of values between stores is not permitted except in some special cases (e.g. when two stores are subsequent and involve the same block)[2]. In contrast, in the memory-side organization, each bbPB entry corresponds to a data block whose value is changed by a store. Because bbPB entries are already in the persistence domain, stores to the same block can be coalesced regardless of the ordering of such stores. Furthermore, ordering is not necessary as store values have reached the persistence domain in bbPB entries. Entries in bbPB can also drain out of order to NVMM, making various optimizations possible, for example, one that minimizes NVMM writes. By allowing store reordering and coalescing, the memory-side organization conveys substantial advantages in requiring fewer bbPB entries to perform well, and in reducing writes to NVMM. Furthermore, the memory-side organization also simplifies cache coherence: since bbPB is at the memory side, it is not directly involved with cache coherence the way L1D or L2 caches are.

The two approaches also differ in handling a load from the core. In the processor-side approach, a load must check the cache hierarchy and bbPB to find the data block. In most cases, the block will be found in the caches instead of the bbPB, but in rare cases, the block may have been evicted from the caches while still residing in the bbPB. In such cases, bbPB supplies the block to the core. Handling a load in the memory-side approach is more complicated. A load first accesses the cache hierarchy. If it misses in the hierarchy (i.e. last level cache (LLC) miss), the block may reside in the memory (NVMM or WPQ) or the bbPB, so both need to be checked. The (MC) may need to inquire the bbPB of each core to find potentially the latest/valid value of the block. Alternatively, to avoid broadcast, a bbPB directory may be kept at the MC to track which bbPB may hold a valid copy of the block. The need to broadcast or keep a directory is a substantial drawback of the memory-side approach.

Thus, for our BBB design, we choose the memory-side approach. However, there is a drawback of the memory-side approach. We note that when the LLC misses, the missed block may still be in bbPB pending to be drained to persistent memory. The memory has a stale copy of the block so the missed block must be located from the right bbPB. To locate the block, a broadcast to all bbPBs in all cores may be needed; or if a directory for bbPBs is kept, only select bbPBs need to be inquired. However, a broadcast is not scalable, but keeping directory information updated requires a complex protocol mechanism as various protocol races could occur. To avoid such a problem, we require that the LLC be *dirty-inclusive* of bbPBs, i.e. any bbPB block must have a corresponding dirty block in the LLC. Being dirty-inclusive, an LLC miss is guaranteed not to find a block in a bbPB, hence eliminating the need to check bbPB on LLC misses. Enforcing inclusion is simple. When a dirty LLC block is evicted, a *forced drain* message is sent to all bbPBs (Figure 5(b)), akin to back invalidation being sent to the L2 and L1 caches. If a bbPB has such a block, it drains the block before responding with acknowledgment.

A dirty block may be drained from bbPB as well as be written back from the LLC. While it is correct to let both occur, for write endurance reason we should avoid redundant write back from the LLC. To quickly identify dirty blocks that should not be written back, we add a bit to each cache block to annotate a block that is holding persistent data, similar to the one used in [50]. When such a block is evicted from the LLC, it is not written back to NVMM. Since a dirty persistent block in LLC has or had a corresponding bbPB block, the value can be considered to have been written back to memory.

C. Handling Relaxed Memory Consistency Models

In an earlier discussion, we described the PoV/PoP gap. There is a subtle issue here related to relaxed memory consistency models. For example, with release consistency, PoV is defined only for and with regard to release synchronization memory instructions but undefined/unordered for regular stores. This creates an ambiguous aspect as to whether PoP for stores should be ordered or left unspecified as the PoV. To guide our choice, we note that PoV is only applicable to multi-threaded applications as it governs when a store from one thread is seen by others, whereas memory persistency applies even to single-threaded (sequential) applications. We believe that the latter requires persist ordering to be defined even for relaxed consistency models. Hence, we propose PoP to follow the semantics of program-order for persisting stores.

A challenge to achieving program-order persistency for relaxed consistency models is that while stores are committed in program order, they do not go to the L1D in program order. For example, if an older store misses in L1D, a younger store that hits the cache is permitted to write its value in the L1D. If an update to bbPB and L1D coincides, we cannot guarantee program order to PoP. To solve this, for relaxed consistency models, we also battery-back the store buffer (SB) (Figure 4). In this design, PoP is achieved when a committed store is allocated in the SB, earlier than PoV which is at the L1D. This requirement for sequential programs is equally needed when using NVCache or eADR, as stores may also write to the cache out of program order. This design adds a small cost to the battery but allows BBB to guarantee program order persistency without requiring the programmer to use persist barriers and without incurring persistency stalls. When a crash happens, the content of the SB will be drained directly to the WPQ (similar to non-temporal stores [3]) after completely draining the content of the corresponding bbPB. This guarantees the per-core program order to be maintained.

[2]An additional coalescing opportunity is possible if epoch persistency is considered: stores within an epoch may be coalesced. However, with BBB we are targeting strict persistency.

D. BBB Design Invariants

BBB design requires the following invariants to be kept to guarantee correct execution and crash recovery:

1) Persisting stores enter the persistency domain in program order. The persistency domain includes bbPB (sequential consistency and total store ordering), or additionally includes store buffer (more relaxed consistency models).[3]
2) Battery consists of sufficient energy to drain bbPB and WPQ (plus store buffers in some cases) to the NVMM in the event of power loss.
3) A store is not made visible to other cores/threads until it becomes persistent.
4) LLC or L2 caches are inclusive of bbPBs, and a block only resides in at most one bbPB.

To meet Invariant 1, a store is allocated a bbPB entry after all older stores have been allocated and written to the cache. If bbPB is full, some entries are drained to free them up. To avoid performance degradation from full bbPB, bbPB needs to be sized sufficiently. If the bbPB already has the block, the new store value will be coalesced with it.

Invariant 2 is ensured by having a battery with sufficient energy to drain bbPB to memory, and thus guranteeing that any allocated bbPB entry will eventually reach the NVMM. This includes in-flight inter-cores packets between bbPBs.

Invariant 3 is common in persistency studies. Violating it may result in a first store from a first thread that has not persisted to become visible to a second thread which then persists a second store that depends on the first store. If a crash occurs, the threads disagree on the persistent state of the first store. To meet Invariant 3, the L1D cache ensures that it has obtained the block in the coherence state that allows the store (i.e. M state), before the store writes to the L1D cache and allocated in the bbPB.

Invariant 4 was partly discussed in Section III-B and the rest will be discussed more below in Section III-E.

E. Cache Coherence Interaction

bbPBs have two unique characteristics. Despite being logically located at the memory side, each core has its own bbPB, and hence if not carefully designed, a block may potentially exist in multiple bbPB and suffer from coherence issues. In order to avoid that, Invariant 4 requires that a block resides in at most one bbPB. The invariant ensures that a block is drained only once from bbPB to NVMM with the latest value, and avoids dealing with coherence between copies at multiple bbPBs. Another unique characteristic of bbPB is that it is located close to the core, and a persisting store needs to allocate an entry as it writes its value to the L1D. To enforce Invariant 4, a writing core cannot just allocate a new entry for a block if the block resides in another bbPB. The block must be removed from the other bbPB and retrieved to the writing core's bbPB.

There are two issues that we need to deal with to support Invariant 4. First, a bbPB must be notified of any relevant external invalidation or intervention request made by another core. This is not as simple as it sounds because the LLC does not keep a bbPB directory; it only keeps directory for per-core L2 caches. Hence, when a core wants to write to a block, it does not know which bbPB to send the invalidation to. To simplify this, we enforce bbPB L2 inclusion, meaning that for each block in bbPB, the same block must also exist in the L2 cache. L2 inclusion provides substantial benefits because the LLC keeps L2 directory, hence by sending invalidation to the sharer L2 caches (which then send back invalidation to their respective bbPBs), it is guaranteed that the bbPB containing the block will be notified as well. No new directory information is needed in the LLC.

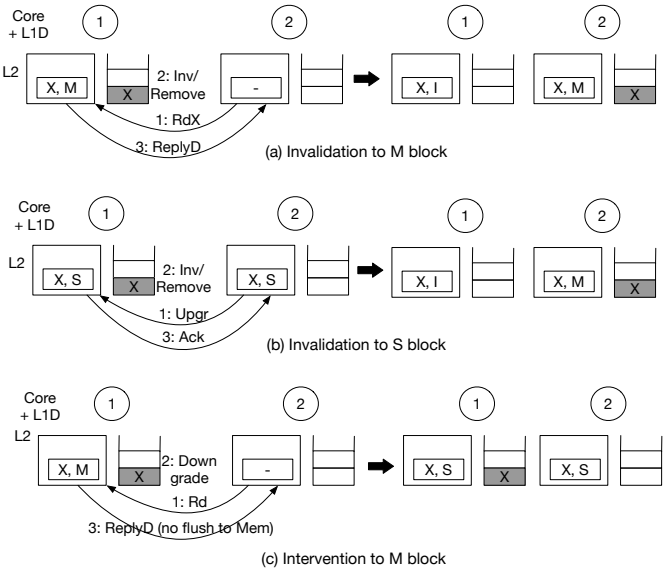

Fig. 6: Illustrating how BBB handles main cache coherence cases with data in bbPBs. Terms follow from [83].

Another issue is whether to drain the block from bbPB when an invalidation/intervention is received by a bbPB. If the block is drained, Invariant 4 is enforced as the block is removed from the current bbPB so that the new bbPB can allocate it. However, draining delays the acknowledgement or reply to the invalidation/intervention until draining is complete, and it incurs an additional write to NVMM which reduces write endurance. Thus, we choose not to drain the block. Instead, when an external request is received, the block is moved to the requesting core. The requesting core is now responsible for draining this block to the NVMM. Note that the that energy source is sized to provide sufficient energy to complete any in-flight packets in the event of crash. Therefore, it is guaranteed that no updates will be lost due to the inter-core movements of cache blocks. This requirement is equally needed for eADR.

Figure 6 illustrates the main coherence scenarios. Two cores are illustrated with the L2 cache and the bbPB is shown for each core. A block X and its initial state (assuming MESI protocol) in the L2 cache of Core1 are shown. It receives an external request from Core2. For example (a), the block

[3]This invariant is equally needed for eADR or any NVCache solution.

TABLE II: Illustrating the bbPB actions corresponding to different coherence operations, originating from other cores (remote invalidation/intervention) or from the same core (local read/write). An operation is marked unmodified (UM) if the base MESI protocol applies.

State	In bbPB?	RemoteInv	RemoteInt	LocalRd	LocalWr
M	N	UM	UM	UM	Allocate
	Y	Fig. 6(a)	Fig. 6(c)	UM	Coalesce
E	N	UM	UM	UM	Allocate
	Y	Invalidate	UM	UM	Coalesce
S	N	UM	UM	UM	Allocate
	Y	Fig. 6(b)	UM	UM	Coalesce
I	N	UM	UM	UM	Allocate
	Y	Invalidate	UM	UM	Coalesce

is in Core1's L2 cache (in M state) and in bbPB. The L2 cache at Core1 receives a Read Exclusive request by Core2 and notifies the bbPB. The L2 cache invalidates the block and bbPB removes the block (without draining it). The block is then sent back to Core2, which then installs it in the L2 cache (in M state), allowing it to write to the block and install it in its bbPB. This example illustrates that if a block is written by multiple cores, the block may move between bbPBs but will drain to memory only once.

In example (b), block X is initially shared by both cores. An Upgrade request is received at Core1's L2 which notifies bbPB. As before, the block is invalidated from the L2 cache and removed from bbPB. An acknowledgment is sent to Core2. At this time, Core2 has sufficient state to allow it to write to the block and simultaneously install it in the bbPB. No draining occurs here, either.

Finally, in example (c), the block is in the M state initially and Core1's L2 cache receives a read request from Core2. In response, Core1 downgrades its block from M to S and replies to the request with data. However, the block remains in the original bbPB. With traditional MESI protocol, the block will be written back to memory because the resulting state S indicates that the block must be clean in the cache. However, our memory-side approach here allows an optimization. Since bbPB is in the persistence domain and can be considered as an extension of the main memory, it is as if the M block had already been written back to memory. Hence, write back to memory is skipped, resulting in bandwidth saving to the LLC.

In conclusion, the modifications to the cache coherence protocol are minor. No additional delay is added to the critical path of cache coherence transactions. Furthermore, our BBB approach allows bbPB to minimize the number of writes to memory, both for bbPB draining, as well as for writeback from the L2 cache. Table II illustrates full coherence cases and the corresponding bbPB operations.

F. Other Issues

a) bbPB draining policy: Another important design issue is regarding when and how to drain bbPB to NVMM. Regarding the *when* question, since blocks in bbPB are already in the persistence domain and there is sufficient energy to drain them to memory, in theory, they can stay in the bbPB for coalescing. Draining bbPB too early reduces opportunities to coalesce multiple stores, which decreases both performance and write endurance. On the other hand, draining too late increases the chance of the bbPB being full when a burst of persisting stores needs new entries allocated, resulting in performance degradation. Hence, an important principle of optimization is to keep bbPB as full as possible while keeping the probability of full bbPB low. To achieve this balance, we define a *draining occupancy threshold*. bbPB does not drain blocks except when its occupancy reaches the threshold, at which time draining is initiated until the occupancy decreases below the threshold. For example, we found 75% threshold to work well for 32-entry bbPB. A similar optimization is applied to memory controller WPQ [34], [35].

Regarding the *how* question, we apply a first come first served (FCFS) draining policy; the oldest block allocated in the bbPB is chosen to drain first. While other policies are possible, e.g. draining blocks based on the prediction for future writes, we leave them for future work.

b) Hardware cost of BBB: Assuming bbPB having 16-32 entries (more on the choice in Section V), the total size of bbPB will be about 1-2KB per core. Each bbPB entry contains a 64-byte data block plus up to 8-byte meta-data that contains the physical block address, and a few bits for status. The physical address is used to avoid accessing TLB when bbPB is drained, or when the L2 cache sends back invalidation to the bbPB.

c) Context switch: Because bbPB holds the block's physical address, there is no cache block address aliasing problem between multiple processes. No draining or state saving is needed on context switch.

IV. Methodology

TABLE III: The simulated system configuration.

Component	Configuration
Processor	8 cores, OoO, 2GHz, 8-wide issue/retire ROB: 192, fetchQ/issueQ/LSQ: 32/32/32
L1I and L1D	private, 128kB, 8-way, 64B, 2 cycles
L2	shared, 1MB, 8-way, 64B, 11 cycles
DRAM	8GB, 55ns read/write
NVMM	8GB, 150ns read, 500ns write (ADR)
bbPB	32 entries per core, drain threshold 75%

A. Simulation configuration

We evaluate BBB using a multicore processor model built on *gem5* simulator [15], with parameters shown in Table III. The machine consists of a hybrid DRAM/NVM main memory, each type being 8GB and having a separate MC. The NVMM MC is in persistence domain and is battery-backed (ADR). The NVMM read and writes latencies are 150ns and 500ns, respectively, which are higher than DRAM latencies, in line with prior studies [14], [20], [47], [55], [86]. We use Arm 64-bit instruction set architecture (ISA). Our simulation models an Arm-based mobile phone with an 8-core processor, each core has an 8-wide out-of-order pipeline. L1 caches are private per

core, while the L2 is shared. Coherence between L1 caches rely on directory-based MESI protocol.

TABLE IV: Summary of the evaluated workloads along with their descriptions and the percentage of the persistent stores (%P-Stores) to the total stores in the workload.

Workload	Description	%P-Stores
rtree	1 million-node rtree insertion	15.5%
ctree	1 million-node ctree insertion	18.9%
hashmap	1 million-node hashmap insertion	6.0%
mutate[NC/C]	modify in 1 million-element array	23.8%
swap[NC/C]	swap in 1 million-element array	23.8%

B. Workload Description

To evaluate battery-backed persist buffers (bbPB) size requirements in BBB, we designed the workloads listed in Table IV. These workloads are chosen to generate significant persist traffic.

Among these workloads rtree, btree, and hashmap workloads maintain a 1 million-node data structure that is allocated in the persistent space, and the workload performs random insertions to the data structure. This generates persistent writes that need to be allocated at the bbPB. Similarly, array-mutate and array-swap perform random mutate and swap operations respectively, on a 1 million-element array. NC or C after the array operation's name (e.g. mutateNC vs mutateC) stands for "Non-Conflicting" or "Conflicting", respectively. This indicates whether each thread performs updates on a separate region of the array (hence non-conflicting), or conflicts are allowed between threads. Each workload runs with 8 threads on 8 cores.

We designed the workloads to exert maximum pressure on the bbPB. They perform back-to-back persistent writes with little other computation. In contrast, real-world workloads typically perform additional computations to generate the data to be persisted. Thus, our analyses on size of bbPB required for good performance represents the worst-case end point for the workloads we studied.

For all of these workloads, we evaluate BBB normalized to eADR which serves as the base case. eADR represents the optimal case for performance overheads and number of writes to NVMM, hence it performs as good as a system without any persistency in mind. On average, the simulated window reports the timing of 250 million instructions, after 200 million instructions for warm-up.

C. Methodology for Evaluating Draining Cost

eADR draining cost depends on the cache hierarchy and number of cores. We evaluate the cost based on two types of systems with differing number of cores and cache hierarchy: a server class and a mobile class system, as shown in Table V. The server class system is based on the specifications of Intel Xeon Platinum 9222 [25], [94], while the the mobile class system are based on the Arm-based iPhone 11 specifications [26], [30], [36]. Most notably, the total cache size for the system is 107MB and 8.75MB for the server and the mobile class system, respectively.

TABLE V: Systems used to evaluate the draining costs

Component	Mobile Class	Server Class
Number of cores	6	32
L1 cache size	6 x 128kB	32 x 32 kB
L2 cache size	1 x 8MB	32 x 1 MB
L3 cache size	N/A	2 x 35.75 MB
Memory channels	2	12

To compare the draining cost of BBB and eADR, we focus on (1) the energy needed at the time of the crash, which determines the size, lifetime, and system footprint of the battery, and (2) the time needed to perform the draining, which is affected by the amount of data to drain and non-volatile main memory (NVMM) write bandwidth. Such time impacts the turn-round time after a crash, and thus the responsiveness of the system. It can also result in further energy overheads if other parts of the system (e.g. the core) need to remain alive during draining.

Estimating draining energy. On a crash, data in bbPB (or in caches for eADR) is accessed and then moved to NVMM. We assume that caches in eADR and bbPB in BBB are SRAM. The energy needed to access data in such SRAM cells is estimated to be about 1pJ/Byte [63]. However, this is very small compared to the energy needed for data movement. The energy required for data movement is a lot harder to calculate.

Our estimations of the energy cost for data movement are based on the results from the work done by Dhinakaran et al [65], which looked into the energy cost of data movement across several levels in the memory hierarchy. The energy consumption per memory operation was measured using an external power meter while executing carefully designed micro-benchmarks. These micro-benchmarks were used to isolate and observe the energy needed solely for data movement and minimize the effect of out-of-order execution and other architectural optimizations. More specifically:

1) To calculate the cost of data movement between the processor and a targeted level in the memory hierarchy (e.g. L2 cache), these micro-benchmarks operate on an allocated data that is chosen such that it has a memory footprint to not fit in any of the cache levels above the targeted level.
2) The average memory latency and the cache miss rates were continuously monitored to validate that the micro-benchmarks are accessing the targeted level of the memory hierarchy.
3) The micro-benchmarks were designed to minimize the impact of other operations, not related to memory accesses.
4) To isolate the compiler optimizations' impact on the micro-benchmarks, all the assembly codes were manually validated to guarantee the expected behavior.

These experiments provided the energy needed to move the data between the processor's registers and any level in the

memory hierarchy. Finally, the difference between these results can be used to calculate the energy cost of moving data between different levels in the memory hierarchy.

Table VI shows the estimated energy needed for draining data from different cache levels to NVMM. The numbers are derived from [65], with some adaptation: (1) As the analysis in [65] only reports data movement in the direction from memory to caches, we estimate that the energy needed to bring data from a cache to memory (as needed at the crash) is similar to that to bring data from memory to the cache. (2) Since the results reported in [65] are only for a DRAM-based system, we assume those results for our analysis on the energy needed to drain to NVMM. This assumption equally affects eADR and BBB, and thus will not have a notable impact on our comparison between the two schemes. (3) The energy for draining a block from bbPB is estimated from the energy to drain a block from the L1D cache to NVMM. (4) The energy numbers in [65] do not include a 3-level memory hierarchy. Therefore, we assume the draining cost numbers do not increase when adding another cache level, as in the server class system in Table V. This assumption produces an optimistic energy figure for eADR, so in reality, eADR energy cost may be higher than our estimate. Moreover, when reporting eADR draining energy and time costs, we calculate only the energy and time needed to drain dirty blocks, to estimate the *average* energy and time.

The final part of our energy analysis is estimating the battery size. In this analysis, the battery needs to be provisioned with sufficient energy to drain the entire caches, in case all blocks in the caches are dirty. This is important because missing to drain even one dirty cache block may result in inconsistent persistent data that cannot be recovered. We chose the smallest battery size that is capable of storing the required energy. Different battery technologies have different energy densities (i.e. the amount of energy stored per volume unit). We looked into two main battery technologies (SuperCap [98] and Li-thin [67]), which have energy density of (10^{-4} and 10^{-2}) Wh cm^{-3}, respectively [93].

TABLE VI: Estimated energy costs of different operations for draining eADR or BBB at the moment of crash.

Operation	Energy Cost
Accessing Data from SRAM	1pJ/Byte
Moving data from L1D to NVMM	11.839nJ/Byte
Moving data from bbPB to NVMM	11.839nJ/Byte
Moving data from L2 to NVMM	11.228nJ/Byte
Moving data from L3 to NVMM	11.228nJ/Byte

Estimating draining time. For this part, we rely on the reported NVMM bandwidth and latencies [41]. As the draining happens at crash with no other traffic present, we assume that the entire NVMM bandwidth will be dedicated for draining. NVMM bandwidth also depends on the number of memory channels for each system as described in Table V.

V. Evaluation

We first discuss the most important aspect of BBB: its draining cost, in comparison to eADR (Section V-A). Then we discuss BBB performance and write overheads (Section V-B and V-C). Finally, we present the sensitivity study of BBB design (Section V-D).

A. Draining Cost Comparison

Table VII presents the average energy needed to drain data from caches (for eADR) and from bbPB (our BBB approach, 32 entries), based on the cost model we discussed in Section IV-C. We give eADR optimistic estimates with several assumptions. First, we assume eADR only drains dirty cache blocks to memory. For the workloads evaluated, on average 44.9% of blocks are dirty in the cache hierarchy, similar to figure obtained by Garcia et al [31]. Second, we assume dirty blocks are identified using a hardware finite state machine that is power efficient, and consumes zero energy overheads. Moreover, we don't include the static energy cost for eADR. In contrast, we note that BBB does not require cache accesses for dirty block identification. Furthermore, caches do not need to be powered during the draining process (hence no static energy consumption). Finally, we assume that at the time of failure, the battery-backed persist buffers (bbPB) are full and all entries need to be drained, representing the worst case for BBB.

As shown in Table VII, despite optimistic estimates, using eADR costs 46.5 mJ and 550 mJ to drain for the mobile and server class systems, respectively. Not surprisingly, the mobile class system has smaller caches hence draining energy is smaller than in the server class system. Despite more realistic estimates, BBB costs only 145 µJ and 775 µJ, respectively, which are 320× and 709× more efficient than eADR, respectively. BBB's energy cost is between two to three orders of magnitude smaller than eADR.

TABLE VII: Estimated draining energy cost for BBB vs. eADR (dirty blocks only).

System	Draining Energy		**Normalized to BBB (X)**	
	eADR	BBB	**eADR**	**BBB**
Mobile Class	46.5 mJ	145 µJ	**320×**	**1**
Server Class	550 mJ	775 µJ	**709×**	**1**

Table VIII presents the average time needed to drain data from caches (for eADR) and from bbPB (our BBB approach). eADR takes 0.8 ms and 1.8 ms to drain for mobile class and the server class system, respectively. In contrast, BBB takes only 2.6 µs (307× faster) and 2.4 µs (750× faster), respectively, which represent two to three orders of magnitude improvement.

eADR and our BBB need energy source to drain. We estimate two energy source types: super capacitors (SuperCap) [98], and lithium thin-film batteries (Li-thin) [67], by applying the analysis from [93] (as discussed in Section IV-C), while using the energy values from Table VII. Table IX shows the estimates for only the active material needed for the battery, excluding packaging and other aspects. As shown in

TABLE VIII: Estimated draining time for BBB vs. eADR (dirty blocks only).

System	Draining Time		**Normalized to BBB (X)**	
	eADR	BBB	**eADR**	**BBB**
Mobile Class	0.8 ms	2.6 µs	**307**×	**1**
Server Class	1.8 ms	2.4 µs	**750**×	**1**

column group (a), eADR in the mobile class system would require an energy source of 2.9×10^3 mm^3 and 30 mm^3 for SuperCap and Li-thin technologies, respectively. In contrast, BBB requires only 4.1 mm^3 and 0.04 mm^3, respectively. The server class system shows a similar trend with energy source of 34×10^3 mm^3 and 300 mm^3 for SuperCap and Li-thin technologies, respectively, for eADR. In contrast, BBB requires only 21.6 mm^3 and 0.21 mm^3 volume, respectively.

To put it in perspective, the last two columns convert size/volume to the footprint area of a typical core used in the mobile class system (i.e. 2.61 mm^2) [30]. Although we don't necessarily envision that this energy is going to be provided through introducing a new battery, we use the comparison to a mobile core's size to help visualize the energy source comparison between BBB and eADR. To simplify converting battery volume to area, we assume cubic battery shape and infer the footprint area from the volume. The areas needed for eADR batteries are substantial: $77\times$ and $404\times$ the size of a core for the mobile class and the server class systems, respectively, when using SuperCap. Even when using Li-thin, which is more space efficient, the area is still large: $3.6\times$ and $18.7\times$ the size of the core, for the mobile class and server class systems, respectively. In contrast, BBB requires a much smaller size. Even with SuperCap, the area needed is 97.2% and 296% the size of a core for the mobile and the server class, respectively. It becomes even smaller with Li-thin: 4.5% and 13.7% the size of a core, respectively. Overall, the battery volume for BBB is between $707 - 1,574\times$ smaller, while the area for BBB is between $79 - 137\times$ smaller than eADR. Moreover, Table X looks into the battery size when varying the number of entries in bbPB and shows that even with bbPB size of 1024 entries, BBB is $22 - 49\times$ cheaper than eADR.

TABLE IX: Estimates of the size of the energy source needed to implement BBB and eADR (a). In addition, to showing the needed footprint to be occupied by the energy source as a ratio to the area of the mobile class system's core (b).

Battery		(a) Size/Volume (mm^3)		(b) Ratio to core area (%)	
		SuperCap	Li-thin	SuperCap	Li-thin
Mobile	eADR	2.9×10^3	30	7,746% ($\sim 77\times$)	359.5% ($\sim 3.6\times$)
	BBB	4.1	0.04	97.2%	4.5%
Server	eADR	34×10^3	300	40,363% ($\sim 404\times$)	1,873% ($\sim 18.7\times$)
	BBB	21.6	0.21	296% ($\sim 3\times$)	13.7%

B. Performance Evaluation

The much smaller amount of data that needs draining is the primary reason for BBB's much smaller battery cost than eADR. However, this may also incur performance degradation in BBB when persisting stores come at a faster pace than can be drained by bbPB, causing bbPB to be full and stalls the core. In contrast, eADR does not cause stalls anymore than regular caches would, hence it represents the ideal case.

TABLE X: Battery size (in mm^3) when varying the number of bbPB entries for mobile (M) and server (S) platforms.

bbPB Size		1	4	16	**32**	64	256	1024
SprCap	M	0.12	0.50	2.02	**4.1**	8.1	32.3	129.3
	S	0.7	2.7	10.8	**21.6**	43.1	172.4	689.7
Li-thin	M	0.001	0.005	0.02	**0.04**	0.08	0.3	1.3
	S	0.006	0.026	0.10	**0.21**	0.43	1.7	6.8

Figure 7(a) illustrates the execution time of two versions of BBB with differing bbPB entries (32 and 1024) and eADR, normalized to eADR, for all workloads tested. As shown in the figure, 32-entry BBB performs worse than eADR by only about 1% on average, and 2.8% in the worst case. The main source of performance overhead is when a core's bbPB is full when the core attempts to persist a store, and coalescing cannot be done because the block being written is not in bbPB. In this case, the store stalls the core until some blocks in bbPB have been drained and free up some entries. With 1024 entries, BBB achieves nearly identical performance with eADR, at the cost of a larger battery compared to 32-entry BBB.

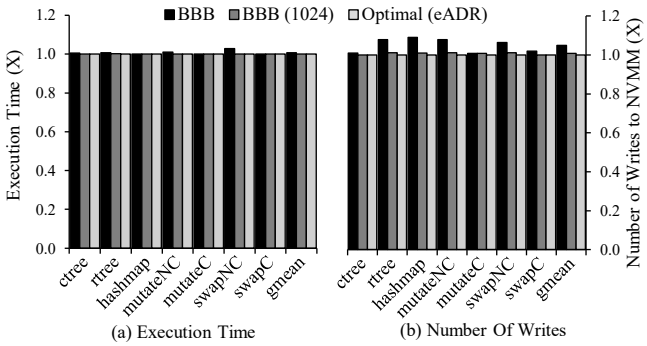

Fig. 7: Execution time and number of writes to NVMM for BBB with 32 entries (first bar), with 1024 entries (second bar), and eADR (third bar), normalized to eADR.

As discussed in Section IV, our workloads were designed to generate back-to-back writes to the persistent domain, which stresses persist buffers. However, the nature of the data structures in the workloads still creates a difference in the time needed to perform the operation and thus creates a difference in the frequency of generating persists by the core. This is why some workloads (e.g. swapNC) might incur relatively higher delays due to the very short time between two subsequent persists.

C. NVMM Writes Evaluation

Due to the limited write endurance of NVMM, the number of writes to NVMM is also an important metric. If we had used the *processor-side* approach, almost every persisting store must go to the bbPB and drain to the NVMM. With the *memory-side* approach, stores are coalesced in bbPB, and the number

of NVMM writes depends on the number of block draining needed to free up entries in bbPB. Hence, we expect the number of bbPB entries to determine the number of NVMM writes. Also, as discussed in Section III, dirty writebacks from the LLC to the NVMM in our BBB approach are silently dropped to avoid redundant writes to the NVMM.

Figure 7(b) compares the number of NVMM writes for BBB with 32 and 1024 entries, with eADR, normalized to eADR. eADR represents the optimal case because eADR does not introduce any new writes to the NVMM due to persistency ordering. The figure shows that even 32-entry bbPB in BBB captures the majority of the coalescing that happen in eADR; it only adds an average of 4.9% writes to NVMM (ranging from $1 - 7.9\%$) to eADR. This overhead decreases to less than 1% if BBB uses 1024-entry bbPB, since the larger buffer provides more room to hold blocks and captures most coalescing opportunities. This result illustrates the effectiveness of the memory-side approach in coalescing stores in the bbPB, because bbPB is in the persistence domain. In contrast, traditional persist buffers prevent most coalescing because it would result in persistency ordering violation, so the number of writes would be much higher.

We also measured the number of writes to NVMM using the processor-side approach, and found that on average, there are $2.8\times$ more writes to NVMM than eADR. This is because there are not many coalescing opportunities, while the memory-side approach is effective in performing coalescing.

D. Sensitivity Studies

To obtain deeper insights into BBB, we vary the bbPB size from 1 entry up to 1024 entries. Figure 8 shows the number of times persisting stores are rejected at the bbPB because it is full (a), execution time overhead (b), and number of bbPB drains to memory (c). All figures are normalized to the 1-entry bbPB case. Note that the y-axes starts at -0.2, so that near zero values are visible in the figures.

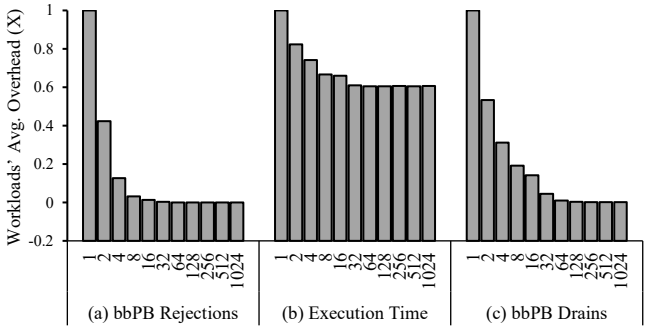

Fig. 8: Sensitivity study showing the average (i.e. geomean) impact of varying the bbPB size on: The number of persist request being rejected due to full bbPB (a), The execution time comparison (b), and the number bbPB drains to the NVMM (c). All are normalized (X) to the case with bbPB size of 1 (leftmost column) in each group.

As shown in the figure, bbPB rejection count decreases quickly when increasing the bbPB size, reaching nearly zero with 16-32 entries. These findings are consistent with the impact on execution time, which stops decreasing with 32 entries. The bbPB drains overhead reaches near zero with 64 entries, but 32 entries are not far behind. This represents the amount of coalescing that was achieved at the bbPB and will be translated into a reduction of the number of writes to the NVMM. Thus, 32-entry bbPB (our default configuration) is the smallest size that shows very close results compared to eADR, beyond which we start to have diminishing returns. This buffer size might be higher than what was used in prior works (e.g. 8 entries in [50]). However, we decided to conservatively choose a relatively larger buffer for the following reasons: (1) We chose the size to conduct the energy comparison between BBB and eADR. Therefore, we chose the smallest size that shows almost no performance degradation (about 1%). Prior works showed acceptable, yet higher, degradation when using smaller sizes, which is consistent with the result we report in Figure 8. (2) Our evaluated workloads were chosen to represent the worst-case and stress the design by generating back-to-back persists. Therefore we expect to have larger buffers. In general, the choice of bbPB size is a design decision based on the trade-off between the energy budget and the desired performance.

VI. Related Work

NVM has received significant research activities in recent years. Past work has examined various aspects of NVM, including memory organization (e.g., [9], [79]), abstraction (e.g., [84]), checkpointing (e.g., [7], [27]), memory safety (e.g., [8], [11], [95], [96]), secure execution environment (e.g., [12], [29]), extending life time (e.g., [6], [21], [22], [70], [72]), and persistency acceleration (e.g., [81], [82]). The above list is a small subset of examples of work in NVM research. From here on, we will expand on papers that are most immediately related to our work.

Persistency models. Persist barriers enable epochs to be defined in BPFS [23]. Pelley et al. defined and formalized memory persistency models including strict, epoch, and strand persistency [68], in the order of increasing performance and decreasing ordering constraints. Persist ordering can be completely relaxed using lazy persistency, as long as persistent state integrity can be checked using checksums [5], [13]. Persist barrier variants that only ensure ordering but not synchronous to instruction execution were presented in [62]. Persistency model is also moving up, being considered at the programming language level [49].

Persist buffers. Volatile persist buffers were introduced in DPO [50] and HOPS [62] to enable buffered persistency models, where stores are temporarily held until drained to memory. In contrast to DPO and HOPS, BBB's persist buffers (bbPB) differ in the following aspects: (1) *battery-backed*, (2) part of the persistence domain, and (3) logically memory-side. These key differences lead to unique characteristics: (1) stores in bbPB can be freely coalesced and reordered, (2) strict persistency is automatically achieved without flushes and fences.

eADR. Persistency domain initially consisted of only the NVM, and has expanded over time. Initially, persisting a

store requires flush, fence, and another instruction *pcommit* which flushes MC write pending queue (WPQ) to NVMM. With ADR, Intel adds WPQ to the persistency domain, thus deprecating pcommit. Capacitor or battery is needed to provide WPQ's flush-on-fail capability. More recently, Intel hinted that eADR will make it into production [78]. eADR [77] adds the entire cache hierarchy to the persistence domain. Because PoV/PoP gap is closed, flush and fence instructions are generally no longer necessary with eADR. However, eADR requires battery to provide flush-on-fail for the entire cache hierarchy instead of only the bbPB in BBB. Table XI summarizes this comparison between eADR and BBB.

TABLE XI: Summary of the comparison between eADR and BBB regarding the hardware/integration costs.

Aspect	eADR	BBB
Processor modifications	None	(1) Adding the bbPBs (2) Minor coherence changes
Draining energy cost	Very High	Low
Time needed to drain	Very High	Low
Drive energy to targeted components	Needed	Needed

Failure atomicity. Persistency programming assumes a certain persistency model and on top of that, relies on support for failure atomic code regions. For sequential programs, libraries and language extensions have been implemented to provide transaction-based failure atomicity [2], [17], [85]. Automatic transformation of data structures to persistent memory has been proposed too [53], [60]. For concurrent programs, compiler-based automatic instrumentation has been proposed to transform lock-based concurrent programs for persistent memory [19], [33]. Other works propose hardware and software primitives to aid porting lock-free data structures to persistent memory [66], [91]. Many studies add durability to transaction-based concurrent programs without changing application source code [32], [44], [92], while others add durability to software transactions [24], [75], [86]. Finally, checkpointing and system-level solutions were also used to achieve failure atomicity [45], [59], [76]. Overall, the aforementioned works provide mechanisms for achieving failure atomic regions (i.e. durable transactions), which is orthogonal the goal of our paper. BBB addresses persist ordering and simplifies ordering-related programming complexity, which provides a property that can be relied on by higher level primitives such as failure atomic regions.

VII. Conclusion

We have proposed Battery-Backed Buffers (BBB), a microarchitectural approach that simplifies the persist ordering aspect of persistency programming by aligning the point of persistency (PoP) with the point of visibility (PoV). We evaluated BBB over several workloads and found that adding 32-entry bbPB per core is sufficient to provide performance comparable to eADR (only 1% slow down and 4.9% extra writes) while requiring $320 - 709\times$ lower draining energy compared to eADR.

References

[1] Intel. an introduction to pmemcheck. [Online]. Available: https://pmem.io/2015/07/17/pmemcheck-basic.html

[2] "Persistent memory development kit (pmdk)." [Online]. Available: https://pmem.io/pmdk/

[3] "Non-temporal store instructions," 2017. [Online]. Available: https://www.felixcloutier.com/x86/movntdq

[4] H. Akinaga and H. Shima, "Resistive Random Access Memory (ReRAM) Based on Metal Oxides," *IEEE Journal*, 2010.

[5] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *ISCA*, 2018.

[6] M. Alshboul, J. Tuck, and Y. Solihin, "Wet: Write efficient loop tiling for non-volatile main memory," in *DAC*, 2020.

[7] M. Alshboul, H. Elnawawy, R. Elkhouly, K. Kimura, J. Tuck, and Y. Solihin, "Efficient checkpointing with recompute scheme for non-volatile main memory," *ACM Trans. Archit. Code Optim.*, 2019.

[8] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers," in *ASPLOS*, 2016.

[9] A. Awad, S. Blagodurov, and Y. Solihin, "Write-aware management of nvm-based memory extensions," in *ICS*, 2016.

[10] A. Awad, B. Kettering, and Y. Solihin, "Non-volatile Memory Host Controller Interface Performance Analysis in High-performance I/O Systems," in *ISPASS*, 2015.

[11] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories," in *ISCA*, 2017.

[12] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-NVM: Persistency for Integrity-Protected and Encrypted Non-Volatile Memories," in *ISCA*, 2019.

[13] A. W. Baskara Yudha, K. Kimura, H. Zhou, and Y. Solihin, "Scalable and fast lazy persistency on gpus," in *IISWC*, 2020.

[14] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, M. Tosi, R. Bez, R. Gastaldi, and G. Casagrande, "An 8Mb Demonstrator for High-density 1.8V Phase-Change Memories," in *VLSIIC*, 2004.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The GEM5 simulator," *ACM SIGARCH Computer Architecture News (CAN)*, 2011.

[16] S. Blanas, "The new bottlenecks of scientific computing," 2020. [Online]: https://www.sigarch.org/from-flops-to-iops-the-new-bottlenecks-of-scientific-computing/

[17] B. Bridge, "Nvm-direct." [Online]: https://github.com/oracle/nvm-direct

[18] M. Carlson. Persistent memory: What developers need to know. [Online]: https://www.snia.org/educational-library/persistent-memory-what-developers-need-know-2018

[19] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, 2014.

[20] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *VLDB Endow.*, Jan. 2015.

[21] J. Chen, G. Venkataramani, and H. H. Huang, "Repram: Re-cycling pram faulty blocks for extended lifetime," in *DSN 2012*, 2012.

[22] J. Chen, Z. Winter, G. Venkataramani, and H. H. Huang, "rpram: Exploring redundancy techniques to improve lifetime of pcm-based main memory," in *PACT*, 2011.

[23] 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 *SOSP*, 2009.

[24] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," in *SPAA*, 2018.

[25] CPU-WORLD, "Intel xeon 9222 specifications." [Online]: http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon\%209222.html

[26] J. Cross, "Inside apple's a13 bionic system-on-chip." [Online]: https://www.macworld.com/article/3442716/inside-apples-a13-bionic-system-on-chip.html

[27] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *PACT*, 2017.

[28] X. Fong, Y. Kim, R. Venkatesan, S. H. Choday, A. Raghunathan, and K. Roy, "Spin-transfer torque memories: Devices, circuits, and systems," *Proceedings of the IEEE*, 2016.

[29] A. Freij, S. Yuan, H. Zhou, and Y. Solihin, "Persist level parallelism: Streamlining integrity tree updates for secure persistent memory," in *MICRO*, 2020.

[30] A. Frumusanu, "The apple iphone 11, 11 pro and 11 pro max review." [Online]: https://www.anandtech.com/show/14892/the-apple-iphone-11-pro-and-max-review/2

[31] A. A. García, R. de Jong, W. Wang, and S. Diestelhorst, "Composing lifetime enhancing techniques for non-volatile main memories," in *MEMSYS*, 2017.

[32] E. Giles, K. Doshi, and P. Varman, "Hardware transactional persistent memory," in *MEMSYS*, 2018.

[33] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *PLDI*, 2018.

[34] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi, "Simulating dram controllers for future system architecture exploration," in *ISPASS*, 2014.

[35] C. Huang, V. Nagarajan, and A. Joshi, "Dca: A dram-cache-aware dram controller," in *SC*, 2016.

[36] A. Inc, "Apple: iphone11 specifications." [Online]: https://www.apple.com/iphone-11/specs/

[37] Intel, "Deprecating the pcommit instruction," 2016. [Online]: https://software.intel.com/blogs/2016/09/12/deprecate-pcommit-instruction

[38] Intel, "Persistent memory programming," 2016, http://pmem.io.

[39] Intel and Micron, "Intel and micron produce breakthrough memory technology," 2015.

[40] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," in *ASPLOS*, 2016.

[41] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[42] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie, "Energy- and endurance-aware design of phase change memory caches," in *DATE*, 2010.

[43] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient Persist Barriers for Multicores," in *Micro*, 2015.

[44] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *ISCA*, 2018.

[45] S. Kannan, A. Gavrilovska, and K. Schwan, "Pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *EuroSys*, 2016.

[46] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. 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 *ISSCC*, 2007.

[47] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: Exploiting nvram in write-ahead logging," in *ASPLOS*, 2016.

[48] Y. Kim, S. R. Lee, D. Lee, C. B. Lee, M. Chang, J. H. Hur, M. Lee, G. Park, C. J. Kim, U. Chung, I. Yoo, and K. Kim, "Bi-layered rram with unlimited endurance and extremely uniform switching," in *2011 Symposium on VLSI Technology - Digest of Technical Papers*, 2011.

[49] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *ISCA*, 2017.

[50] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *MICRO*, 2016.

[51] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an Energy-effcient Main Memory Alternative," in *ISPASS*, 2013.

[52] B. C. Lee, "Phase Change Technology and The Future of Main Memory," *IEEE Micro*, 2010.

[53] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: converting concurrent dram indexes to persistent-memory indexes," in *SOSP*, 2019.

[54] Z. Lin, M. Alshboul, Y. Solihin, and H. Zhou, "Exploring memory persistency models for gpus," in *PACT*, 2019.

[55] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," in *ASPLOS*, 2017.

[56] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan, "Cross-failure bug detection in persistent memory programs," in *ASPLOS*, 2020.

[57] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," in *ASPLOS*, 2019.

[58] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-Ordering Consistency for Persistent Memory," in *ICCD*, 2014.

[59] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx," in *EuroSys*, 2017.

[60] A. Memaripour, J. Izraelevitz, and S. Swanson, "Pronto: Easy and fast persistence for volatile data structures," in *ASPLOS*, 2020.

[61] S. Mittal, J. S. Vetter, and D. Li, "Lastingnvcache: A technique for improving the lifetime of non-volatile caches," in *2014 IEEE Computer Society Annual Symposium on VLSI*, 2014.

[62] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," *SIGPLAN Not.*, 2017.

[63] D. Nayak, D. P. Acharya, and K. Mahapatra, "An improved energy efficient sram cell for access over a wide frequency range," *Solid-State Electronics*, vol. 126, 2016.

[64] K. Oleary. How to detect persistent memory programming errors using intel inspector. [Online]: https://software.intel.com/en-us/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector

[65] D. Pandiyan and C.-J. Wu, "Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms," *IISWC*, 2014.

[66] M. Pavlovic, A. Kogan, V. J. Marathe, and T. Harris, "Brief announcement: Persistent multi-word compare-and-swap," in *PODC*, 2018.

[67] D. Pech, M. Brunet, H. Durou, P. Huang, V. Mochalin, Y. Gogotsi, P.-L. Taberna, and P. Simon, "Ultrahigh-power micrometre-sized supercapacitors based on onion-like carbon," *Nature nanotechnology*, 2010.

[68] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *ISCA*, 2014.

[69] PMEM.io, "Persistent memory programming," 2019. [Online]: https://pmem.io/2019/12/19/performance.html

[70] M. K. Qureshi, "Pay-as-you-go: Low-overhead hard-error correction for phase change memories," in *MICRO*, 2011.

[71] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, *Phase Change Memory: From Devices to Systems*, 2011.

[72] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *MICRO*, 2009.

[73] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, "Persistency semantics of the intel-x86 architecture," *Proc. ACM Program. Lang.*, 2019.

[74] R. Rajachandrasekar, S. Potluri, A. Venkatesh, K. Hamidouche, M. W. ur Rahman, and D. K. D. Panda, "MIC-Check: A Distributed Checkpointing Framework for the Intel Many Integrated Cores Architecture," in *HPDC*, 2014.

[75] P. Ramalhete, A. Correia, P. Felber, and N. Cohen, "Onefile: A wait-free persistent transactional memory," in *DSN*, 2019.

[76] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *MICRO*, 2015.

[77] A. Rudoff, "Persistent memory programming."

[78] A. Rudoff, "Persistent memory programming without all that cache flushing," in SDC, 2020.

[79] M. Saxena and M. M. Swift, "Flashvm: Virtual memory management on flash," in *USENIXATC*, 2010.

[80] S. Scargall, "Persistent memory architecture," in *Programming Persistent Memory*, 2020.

[81] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *MICRO*, 2017.

[82] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *ISCA*, 2017.

[83] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC Computational Science, 2015.

[84] Y. Solihin, "Persistent memory: Abstractions, abstractions, and abstractions," *IEEE Micro, 39(1)*, 2019.

[85] P. Subrahmanyam, "pmem-go." [Online]: https://github.com/jerrinsg/go-pmem
[86] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *ASPLOS*, 2011.
[87] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi, "i2wap: Improving non-volatile cache lifetime by reducing inter- and intra-set write variations," in *HPCA*, 2013.
[88] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck, "Hardware supported persistent object address translation," in *MICRO*, 2017.
[89] T. Wang, S. Sambasivam, and J. Tuck, "Hardware supported permission checks on persistent objects for performance and programmability," in *ISCA*, 2018.
[90] W. Wang and S. Diestelhorst, "Quantify the performance overheads of pmdk," in *MEMSYS*, 2018.
[91] W. Wang and S. Diestelhorst, "Brief announcement: Persistent atomics for implementing durable lock-free data structures for non-volatile memory," in *SPAA*, 2019.
[92] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, "Persistent transactional memory," *IEEE Computer Architecture Letters*, 2014.
[93] Z.-S. Wu, K. Parvez, X. Feng, and K. Müllen, "Graphene-based in-plane micro-supercapacitors with high power and energy densities," *Nature communications*, 2013.
[94] I. Xeon, "Intel® xeon® platinum 9222 processor." [Online]: https://ark.intel.com/content/www/us/en/ark/products/195437/intel-xeon-platinum-9222-processor-71-5m-cache-2-30-ghz.html
[95] Y. Xu, Y. Solihin, and X. Shen, "Hardware-Based Domain Virtualization for Intra-Process Isolation of Persistent Memory Objects," in *ISCA*, 2020.
[96] Y. Xu, Y. Solihin, and X. Shen, "MERR: Improving Security of Persistent Memory Objects via Efficient Memory Exposure Reduction and Randomization," in *ASPLOS*, 2020.
[97] L. Zhang and S. Swanson, "Pangolin: A fault-tolerant persistent memory programming library," in *USENIXATC*, 2019.
[98] Y. Zhu, S. Murali, M. D. Stoller, K. J. Ganesh, W. Cai, P. J. Ferreira, A. Pirkle, R. M. Wallace, K. A. Cychosz, M. Thommes, D. Su, E. A. Stach, and R. S. Ruoff, "Carbon-based supercapacitors produced by activation of graphene," *science*, 2011.