Language-level persistency

Aasheesh Kolli Vaibhav Gogte Ali Saidi Stephan Diestelhorst Peter M. Chen
Satish Narayanasamy Thomas F. Wenisch
University of Michigan ARM
{akolli,vgogte,pmchen,nsatish,twenisch}@umich.edu,{ali.saidi,stephan.diestelhorst}@arm.com

ABSTRACT

The commercial release of byte-addressable persistent memories, such as Intel/Micron 3D XPoint memory, is imminent. Ongoing research has sought mechanisms to allow programmers to implement recoverable data structures in these new main memories. Ensuring recoverability requires programmer control of the order of persistent stores; recent work proposes persistency models *as an extension to memory consistency to specify such ordering. Prior work has considered persistency models at the abstraction of the instruction set architecture. Instead, we argue for extending the language-level memory model to provide guarantees on the order of persistent writes.*

We explore a taxonomy of guarantees a language-level persistency model might provide, considering both atomicity and ordering constraints on groups of persistent stores. Then, we propose and evaluate Acquire-Release Persistency *(ARP), a language-level persistency model for C++11. We describe how to compile code written for ARP to a state-of-the-art ISA-level persistency model. We then consider enhancements to the ISA-level persistency model that can distinguish memory consistency constraints required for proper synchronization but unnecessary for correct recovery. With these optimizations, we show that ARP increases performance by up to 33.2% (19.8% avg.) over coding directly to the baseline ISA-level persistency model for a suite of persistent-write-intensive workloads.*

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Software and its engineering** → **Software notations and tools**;

KEYWORDS

Persistent memories, memory persistency, language-level models

ACM Reference format:
Aasheesh Kolli Vaibhav Gogte Ali Saidi Stephan Diestelhorst Peter M. Chen Satish Narayanasamy Thomas F. Wenisch. 2017. Language-level persistency. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 13 pages.
https://doi.org/10.1145/3079856.3080229

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080229

1 INTRODUCTION

Persistent Memories (PMs), such as Intel's upcoming 3D XPoint memory [22], offer many desirable properties, such as the durability of disk, better density & energy efficiency than DRAM, and DRAM-like performance. These properties have spawned myriad efforts to adopt PM in different areas of computer science, ranging from data structures [23, 51], to software systems [10, 13, 14, 18, 53], to computer architecture [25, 30, 41]. One of the most disruptive potential use cases for PM is to host in-memory recoverable data structures. PMs blur the traditional divide between a byte-addressable, volatile main memory and a block-addressable, persistent storage. This memory allows programmers to directly manipulate recoverable data structures using processor loads and stores, rather than relying on performance-sapping software intermediaries like the operating system and file system [13, 53].

Ensuring the recoverability of data structures requires programmers to have the ability to control the order stores reach PM [5, 14, 18, 41, 59]. With write-back caching, stores may reach PM out of order, compromising data structure recoverability. Existing systems do not provide efficient mechanisms to enforce the order in which stores are written back [5, 29]. Recent work has proposed *persistency models* to provide programmers an interface to control the order persistent stores write to PM [3, 14, 21, 25, 41]. Like prior work, we refer to the act of writing a store durably in PM as a *persist*.

Various persistency models have been proposed, but all of them have been specified at the instruction set architecture (ISA) level. That is, programmers must reason about recovery correctness at the abstraction of assembly instructions, an approach which is error prone and places an unreasonable burden on the programmer. The programmer must invoke ISA-specific mechanisms (via library calls or inline assembly) to ensure persist order, and often must reason carefully about compiler optimizations that may affect the relevant code. Since the ISA mechanisms differ in sometimes subtle ways, it is hard to write portable recoverable programs.

In this paper, we argue for a *language-level persistency model* that provides mechanisms to specify the semantics of accesses to PM (including with respect to program failures) as an integral part of the programming language, just as language-level memory consistency models enable precise specification of the semantics of memory accesses from concurrent threads. A language-level persistency model provides a single, ISA-agnostic framework for reasoning about persistency and can enable portability of recoverable software across language implementations (compiler, runtime, ISA, and hardware). Furthermore, a language-level model prescribes precise requirements on the implementation, allowing implementers to reason about the correctness of compiler and hardware optimizations.

We consider how to specify a persistency model that extends the *data-race-free* (DRF) consistency model [1] that is espoused by

popular high-level programming languages like C++11 and Java. The DRF model is appealing for programmers because DRF guarantees a sequentially consistent (SC) execution for data-race-free programs [7], a guarantee often called "SC for DRF". At the same time, the DRF model admits compiler and hardware optimizations that reorder and optimize memory accesses between (and, in certain cases, across) synchronization. Such reorderings and optimizations are invisible to the programmer, because they cannot be observed without a data race.

One might hope that the simplicity of "SC for DRF" might extend naturally to memory persistency. Unfortunately, DRF is insufficient to define semantics for the PM state that recovery code may observe after a failure. The fundamental problem is that failures, such as operating system crashes, hardware lockups, or power disruptions, may occur *at any time*, and thereby introduce a data race into an otherwise race-free program: loads performed during recovery inherently race with stores before the failure. A failure may interrupt the atomicity of a critical section, exposing a partial (and possibly reordered) set of updates to PM to recovery code.

In this paper, we consider how a language-level persistency model might be specified to provide semantics for the PM state after a failure. In Section 3, we explore a taxonomy of guarantees that a language-level persistency model might provide. Stronger guarantees (e.g., *failure-atomicity* of critical sections) make writing recoverable software easier but impose substantial requirements on the implementation, which entail performance penalties. Weaker guarantees complicate reasoning about recovery, but provide greater implementation freedom and performance. The weaker guarantees relax atomicity of critical sections and instead provide only *ordering* guarantees for individual persists. Ordering individual persists allows synthesis of higher granularities of atomicity via logging.

Based on our taxonomy, in Section 4, we propose a concrete model, Acquire-Release Persistency (ARP), to extend the C++11 memory model. We describe how to compile ARP to an existing ISA-level persistency model [30]. Ideally, the language and ISA persistency models work in concert to enforce only the minimal guarantees required for correct recovery. However, we find that mismatch between ARP and the ISA-level model lead to extra constraints that hamper performance. In Section 5, we propose modifications to the C++11 language, compiler, ISA, and hardware to resolve these mismatches, increasing available persist concurrency and scheduling flexibility. The greater flexibility allows the PM controller to reduce page miss rates, improving application performance.

In summary:

- We make a case for language-level rather than ISA-level persistency models.
- We explore a taxonomy of guarantees that a language-level persistency model might provide.
- We propose acquire-release persistency as an extension to the C++11 memory model. We demonstrate that writing applications to ARP rather than the ISA-level persistency model improves performance by up to 18.5% (8.9% avg.).
- We show that, with small extensions to C++11 and the ISA-level persistency model, we can eliminate further unnecessary persist constraints, leading to speedups of up to 33.2% (19.8% avg.).

2 BACKGROUND

In this section we briefly cover relevant background.

2.1 Memory persistency models

PM technologies can maintain recoverable data structures in main memory. However, to ensure correct recovery, programmers need the ability to control the order in which stores persist. To this end, memory persistency models have been proposed, both in academia [14, 25, 41] and in industry [3, 21]. Persistency models provide ISA-level primitives that programmers can use to communicate the desired order of persists to the hardware. It is the responsibility of the hardware to ensure that the specified order of persists is enforced.

2.2 Delegated persist ordering

Delegated persist ordering (DPO) is an implementation strategy for ISA-level persistency models, like the *relaxed consistency buffered strict persistency* (RCBSP) model proposed in [30]. DPO implementations augment the cache hierarchy to record dependencies among stores to PM and later communicate these stores and the dependencies among them to the PM controller. For relaxed consistency models, recording dependencies among stores involves recording not only the stores themselves but also the fences that order the stores. All recorded stores and fences are orchestrated to drain into the write queue at the PM controller in an order legal with respect to the persistency model. The responsibility of persisting the stores is left with the PM controller. The PM controller is the system component most aware of the device level characteristics of the PM and hence can make the best scheduling decisions (e.g., scheduling read-to-write bus turnarounds [59]).

The effect of draining both stores and fences to the memory controller is that the fences divide the stores in the write queue into *epochs*. The PM controller persists stores respecting the epoch order (i.e., stores within the same epoch may persist concurrently). When more persists fall into an epoch, the memory controller enjoys greater flexibility to make better scheduling decisions [4, 27, 28, 32, 59], resulting in better performance. In Section 5, we show how to leverage the flexibility of the C++11 memory model to increase persists per epoch at the PM controller.

2.3 Language-level persistency models

All memory persistency models proposed to date [3, 14, 21, 25, 30, 41] have been specified at the ISA level. These models vary in the semantics they provide. To use these models, programmers must reason about and annotate their programs with assembly instructions to ensure correct persist order. Whereas the challenge of reasoning using assembly instructions might be mitigated by encapsulating assembly annotations in persistency-model-specific libraries, there is no easy way for programmers to develop portable recoverable software. Moreover, without a precise definition of language-level persistency semantics, otherwise legal compiler optimizations could render data structures unrecoverable. These challenges are reminiscent of the motivation for portable language-level memory consistency models [1]. Similarly, we argue for a language-level persistency model, so that programmers do not have to reason about ISA-specific assembly code while developing recoverable software.

2.4 Failure and recovery

A persistency model imposes requirements on the fault-free execution of a program that writes to PM to ensure that, in the event of a failure, a programmer can rely on some set of guarantees on PM state. These guarantees then make it possible to develop recovery software that can repair data structure inconsistencies caused by interrupted updates. Past work on ISA-level persistency models has focused primarily on power failures (since PM state survives power failure). In this work, we consider fail-stop failures more broadly, e.g., program, run-time and operating system crashes, and hardware failures in addition to power failures. Notably, the trivial solution of providing battery backup to drain in-flight persist operations is not sufficient to tolerate all fail-stop failures. For example, an OS crash might expose that the compiler has reordered two persists and may compromise recovery. (We set aside PM media failures, as orthogonal mechanisms are required to tolerate these, e.g., [16]).

After a failure, we assume the contents of all volatile state (processor registers including program counters, cache contents, volatile memory) as well as incomplete persists are lost, but the contents of persistent memory are retained. Recovery software then examines the persistent data structure, repairing it if necessary, so that normal operation may resume. In some cases, normal operation may be able to resume without any recovery. For example, prior work has demonstrated that wait-free concurrent data structures are inherently recoverable [24, 38]. We discuss the difficulty of writing recovery code under various persistency guarantees in Section 3.

2.5 DRF Consistency

Popular high-level programming languages like C++11 and Java espouse the data-race free (DRF) memory model to enable parallel programming. One of the key advantages of the DRF memory model is that, for DRF programs, it allows programmers to reason about memory access interleaving at the granularity of *synchronization-free regions*, rather than individual accesses. The lack of any data races implies that programmers are assured that the writes in any synchronization-free region will become visible atomically to other threads. Compilers exploit this guarantee to perform optimizations that reorder memory accesses within a region [7], which wouldn't be permissible otherwise.

In addition to its sequentially consistent synchronization operations, C++11 also provides *low-level atomics*, which allow the programmer to label individual synchronization operations with specific memory ordering semantics. A program that uses relaxed atomics has well-defined memory semantics, but loses the "SC for DRF" guarantee. That is, programs with low-level atomics do not necessarily exhibit SC execution [7]. We consider how persistency models might interact with C++11 programs both with and without low-level atomics.

3 DESIGN EXPLORATION

In this section, we explore possible approaches to design a language-level persistency model.

3.1 Atomicity and ordering

A language-level persistency model has to provide programmers with guarantees on two orthogonal properties: (a) the granularity of failure-atomic regions (i.e., persists from one region are committed to PM atomically) and (b) the ordering of these regions. Programmers need both these guarantees to write correct recoverable software. Figure 1 (a) shows the various options that a language may choose to provide for each of these guarantees and places existing academic and industrial proposals for persistent programming within this taxonomy. The granularity of failure atomicity can vary from an individual persist (8-byte atomic writes) to a synchronization free region (code between two synchronization accesses) to an outer critical section (code between the first lock acquired by a thread until the thread holds no locks). It is important to note that if a programmer desires a larger granularity of failure atomicity than what is natively provided by the language, she can achieve it through undo or write-ahead logging mechanisms [29]. Furthermore, the language may guarantee that these atomic units may be ordered sequentially (SC order) or provide a more relaxed ordering mechanism. For example, the language may provide *sequence points* that the programmer can use to break a thread into epochs. Failure-atomic units within an epoch are unordered, but epochs are sequentially ordered ([14, 41]).

3.1.1 DRF Persistency?

One might argue that it is natural to extend the SC for DRF consistency guarantee to recovery code that executes after failure. That is, it would help programmers in writing recovery code to provide a failure-atomicity guarantee for regions of persists. Such a guarantee would hide compiler or hardware memory access reordering from the programmer, and recovery code need only consider memory states that can arise at synchronization points.

However, arbitrary fail-stop failures make such atomicity challenging to enforce. If writes may persist from a synchronization-free region incrementally, recovery code may observe intermediate memory state within the region, breaking the atomicity guarantee that is core to the DRF model: recovery code is not guaranteed to observe sequentially consistent state. Two ways to resolve the conflict between arbitrary failures and DRF are:

- **Enforce atomicity:** Programming languages may demand that the implementation provide a programmer-transparent mechanism to ensure failure-atomicity (e.g., undo logging in the hardware or runtime).
- **Forego atomicity and provide only ordering:** Alternatively, languages may forego guarantees that program regions appear atomic to post-fault recovery code, and instead guarantee only the relative order of persists, much like ISA-level persistency models. (Note that, in fault-free execution, the SC for DRF guarantees still apply).

Since all persists may be externally visible (to recovery code upon failure), a compiler may not introduce spurious persists or elide existing ones. Effectively, persistent variables must be treated like *volatile* variables in C++11. Next, we explore design alternatives and their implications for the programmer, compiler, and implementation.

3.2 A Taxonomy of Persistency Guarantees

We use a running example to highlight how alternative guarantees can be used to ensure recovery correctness. Consider a program with two shared objects, A and B, each with record fields (R) protected by a lock (Fig. 1 (b)). Suppose the correctness requirement is that the fields of each object must be updated atomically with respect to

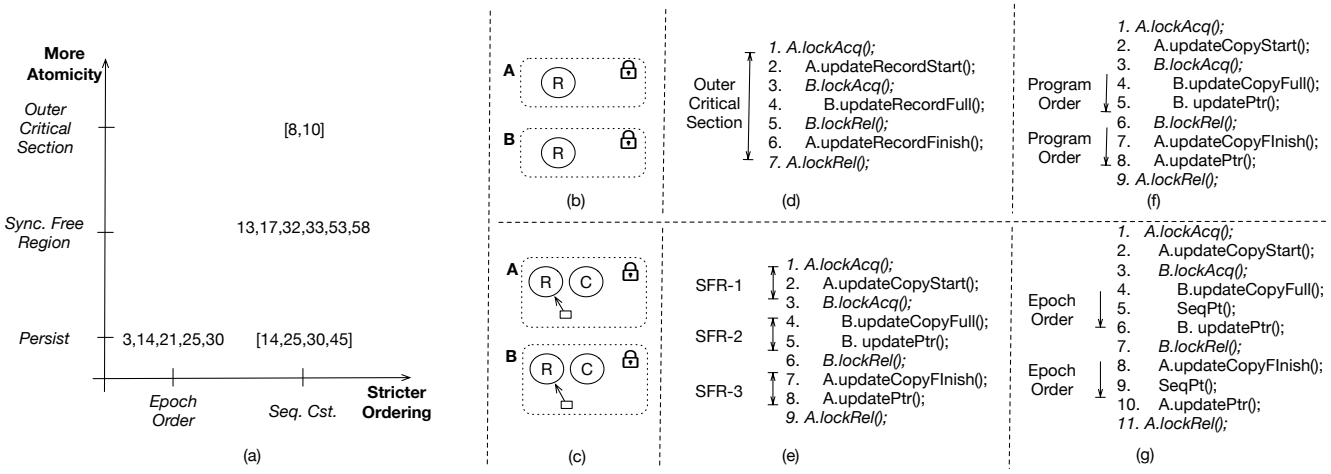

Figure 1: (a) Design space of persistency guarantees explored along two dimensions, atomicity and ordering. (b) Two objects (A,B), each with a record (R) and lock assuming the language provides failure-atomicity of outer critical sections. (c) Two objects (A,B), each with a record (R), a lock, a shadow copy (C), and a pointer to ensure failure-atomicity assuming the language does *not* provide failure-atomicity of outer critical sections. (d) Code and failure-atomic region when the language guarantees sequentially consistent failure-atomic outer critical sections. (e) Code and failure atomic regions when the language guarantees sequentially consistent failure-atomic synchronization free regions. (f) Code and orderings when the language guarantees sequentially consistent persists. (g) Code and orderings when the language guarantees epoch ordered persists.

failures and with respect to other threads. Now, consider code that acquires the lock for object A, starts modifying it, and then must also modify B, which it does within a nested critical section (Fig. 1 (d)). The two locks assure atomicity with respect to concurrent access from other threads in fault-free execution. The persistency model must enable the programmer to write code that can recover to a correct state (i.e., each object to either its initial or final state) in the event of failure.

For languages that do not guarantee the failure-atomicity of the entire update of objects A and B, we provide alternative designs for A and B that rely on shadow logging, shown in Fig. 1 (c). The update is performed on a shadow copy (C) of the object (rather than an in-place update in the object itself). Once the shadow copy has been updated, a pointer is atomically switched to indicate that the copy is committed. For this approach to be correct with respect to recovery, the language must guarantee that the pointer switch persists no earlier than the updates to the shadow copy (assuming appropriate annotations from the programmer). We next consider four different sets of guarantees that a language may provide to enable such recovery. We discuss them in the order of decreasing constraints on persists.

3.2.1 Seq. consistent, failure-atomic outer critical sections

Description: All the persists from an outer critical section (from first lock acquire till no locks are held) are guaranteed by the language implementation to be failure atomic. Further, different outer critical sections must persist in sequentially consistent order.

Example: Fig. 1 (d) shows code which updates both objects in nested critical sections. As the entire outer critical section (from line 1 to 7) is failure-atomic, the condition for correct recovery (each element is individually atomically updated) is trivially met.

Programmability: The idea of sequentially consistent failure-atomic outer critical sections was first explored by Chakrabarti [8, 10]. The central appeal of this guarantee is that, by ensuring failure-atomicity of entire critical sections, the state of persistent memory post-recovery always reflects a state that would have arisen in fault-free execution and when no thread holds a lock. When no locks are held, shared data structures are always in a consistent state. So, no recovery code is needed; the programmer is assured that her data structures are always in a consistent state post-recovery.

Implementation: Chakrabarti [10] provides a software undo-logging mechanism to ensure failure-atomicity of critical sections. Note that the software logging occurs outside of the language's memory model and must be implemented by the runtime system using ISA-level memory persistency.

Compiler optimizations: Since critical sections persist atomically, any compiler optimizations valid within a critical section under fault-free execution remain valid; optimization is unaffected by the persistency guarantee.

Challenges: While failure-atomic critical sections provide an intuitive guarantee, several challenges must be addressed:

(1) *Guarantees for programs without critical sections:* This approach provides no semantics for programs without critical sections (e.g., single-threaded programs). It is unclear how system calls within critical section should be addressed.
(2) *Implementation complexity:* Overlapping critical sections introduce considerable complexity to the logging and log-pruning mechanisms. They may cause cyclic dependencies, which must be carefully resolved [10].
(3) *Large critical sections:* Providing atomicity guarantees over large regions increases the forward progress loss upon a failure. Large and nested critical sections introduce hardware

logging challenges similar to those seen with unbounded transactional memory designs [2].

(4) *Alternatives to logging:* Many data structures can be made recoverable without logging (e.g., wait-free data structures [24, 38]). Furthermore, logging can often be optimized for special cases to improve efficiency (e.g., static transactions [29]). A generic, programmer-transparent logging mechanism will miss these optimization opportunities.

3.2.2 Seq. consistent, failure-atomic synchronization free regions

Description: All persists from a synchronization free region (SFR) are guaranteed to be failure-atomic. Regions must persist in a sequentially consistent order. An SFR is defined as code on the same thread separated by two synchronization accesses, or two system calls, or a synchronization access and a system call [34, 40, 47]. For transaction-based code, the outer critical section is from transaction begin to transaction end. For nested transactions, we assume that inner transactions are flattened into a single outer transaction.

Example: As the modifications to object A span SFRs (due to nested locking), we must use shadow logging to achieve failure-atomicity (Fig. 1 (c)). Fig. 1 (e) shows the code required to ensure that the pointer switch does not persist earlier than the shadow copy update under sequentially consistent failure-atomic SFRs. For object B, since the shadow copy update and pointer update are in the same SFR (SFR-2), the update is failure-atomic. For object A, since the pointer update cannot persist earlier than the partial copy update in SFR-1 (program order of SFRs) or the partial copy update in SFR-3 (failure-atomicity of an SFR), failure-atomicity is preserved.

Programmability: For transaction-based programs or programs without overlapping critical sections, SFRs and critical sections are the same. However, for programs which have overlapping critical sections (as in Fig. 1 (e)), a critical section may span multiple SFRs. For such programs, partially completed critical sections may be visible post-recovery. While developing recovery software, the programmer must be cognizant of this possibility. If failure-atomicity of outer critical sections is desired, the programmer must add roll-back mechanisms for partially completed critical sections.

Implementations: Various logging proposals can provide failure-atomicity for SFRs. However, most focus only on transaction-based code [13, 32, 53, 58]. While transactions simplify logging, they are not general enough to be provided as a language guarantee [8].

Compiler optimizations: Since SFRs persist atomically, optimizations within an SFR remain valid.

Challenges: Several challenges remain under this model:

(1) *Large SFRs:* Large SFRs pose the same challenges as large outer critical sections, as discussed above.

(2) *Alternatives to logging:* As with failure-atomic critical sections, the implementation must provide a generic logging mechanism that will miss data-structure-specific optimization opportunities.

3.2.3 Seq. consistent persists (SCP)

Description: Individual stores persist atomically. All stores persist in sequentially consistent order.

Example: Fig. 1 (f) shows the code required to ensure that the pointer switch does not persist earlier than the shadow copy update under SCP. Since the shadow copy update precedes the pointer switch in program order (lines 4-5 and 7-8 in Fig. 1 (f)), failure-atomicity of the object is preserved.

Programmability: Since only the atomicity of individual persists is guaranteed, the programmer must implement failure-atomicity mechanisms if larger granularities are required. The programmer can rely on the sequentially consistent order of persists while implementing the logging mechanisms.

Implementation: Under SCP, the implementation is no longer required to provide a logging mechanism; it is expected that the programmer will implement mechanisms needed for failure-atomicity in software. Persists drain incrementally to PM, but, the compiler and hardware must ensure that they drain in program order. Under some ISA-level persistency models, stores may need to be flushed individually with explicit instructions [3, 21] or by inserting fence instructions after each store [14, 30]. Hardware can also guarantee SCP via hardware logging [25] or via transparent checkpointing [45].

Compiler optimizations: A consequence of the sequential consistency requirement on stores is that compiler or hardware optimizations that reorder persistent writes are no longer allowed. An implementation may provide atomicity over some regions to allow intra-region reordering, as in speculative consistency implementations [6, 9].

Challenges: While SCP does not require any annotations to ensure persist order, it entails the following challenges:

(1) *In-program logging:* The programmer must implement failure-atomicity mechanisms. However, she is also free to leverage data-structure specific recovery optimizations. Notably, some (e.g., wait-free) data structures require no logging at all [24, 38].

(2) *ISA-level persistency mismatch:* The ISA-level persistency models proposed to date require persistent stores to be flushed individually and fence/barrier instructions to enforce order. For such ISAs, the compiler must insert copious (and performance-sapping) annotations.

(3) *Lost compiler optimizations:* Straight-forward implementation of SCP precludes all compiler and hardware optimizations that reorder writes.

(4) *Performance:* Prior works [14, 41] observe that preserving program order is expensive (due to high PM access latencies) and often unnecessary. Instead they argue for an epoch-based ordering of persists, where programmers use special barrier instructions to indicate required ordering.

3.2.4 Epoch ordered persists (EOP)

Description: This guarantee is derived from ISA-level epoch persistency models [14, 25, 41] proposed in prior research. Special *sequence point* (SP) annotations may be used by a programmer to break a thread into epochs; persists across epochs are ordered, but may be reordered within epochs. Persists on different threads are still governed by synchronization order.

Example: Since the shadow copy update is ordered before the pointer switch via an intermediate SP (lines 5 and 9 in Fig. 1 (g)), the failure-atomicity of each object is ensured.

Programmability: Similar to programming under SCP, programmers may have to implement failure-atomicity mechanisms in software. However, the programmer may no longer rely on program

order, but instead must issue explicit sequence points when ordering guarantees are required, complicating the implementation of recoverable data structures.

Compiler optimizations: The compiler (and hardware) may reorder persists within epochs (e.g., between two sequence points), but may not allow persists to reorder across epochs.

Implementation: Many approaches to implement epoch-persistency models in hardware have been proposed [14, 25, 30]. Any of these satisfy the requirements of EOP.

Challenges: While EOP alleviates many challenges that arise under SCP, some challenges remain:

(1) *In-program logging:* The programmer must use explicit sequence points to ensure recovery correctness rather than simply relying on program order.
(2) *Compiler optimizations:* The compiler may not reorder persists across sequence points.

3.3 Discussion

Each of the four sets of guarantees analyzed in the previous section have their own advantages and disadvantages. Ignoring performance concerns, programmers would clearly want to choose sequentially consistent failure-atomic outer critical sections as the guarantee that languages should provide, as it requires no logging from the programmer. Instead the compiler, runtime or hardware are responsible for providing failure-atomicity of critical sections. However, indications from hardware vendors (e.g., Intel [21], ARM [3]) are that future processors are only going to guarantee the atomicity of individual persists. Because compiler or runtime logging mechanisms [10] required to ensure failure-atomicity must be general, they cannot take advantage of data-structure-specific optimizations (e.g., wait-free recoverable data structures [24], static transactions [29]).

Given that all the other sets of guarantees would require programmers to implement some in-program logging, we argue that the language should provide the most fundamental atomicity guarantee (individual persists); software solutions (e.g., in expert-crafted libraries) for larger atomic regions can be layered on top to reduce programmer burden. In the rest of this paper, we focus on analyzing, designing, and evaluating implementations of SCP and EOP.

4 ACQUIRE-RELEASE PERSISTENCY

We next propose acquire-release persistency (ARP), a persistency model for C++11 based on the EOP approach.

4.1 Definition

We formally define ARP as an ordering relation over memory events—loads and stores on data variables, acquire and release operations on atomic variables—and sequence points. By "thread", we refer to execution contexts—cores or hardware threads. We use the following notation:

- A_x^i: An acquire operation from thread i on atomic variable x
- R_x^i: A release operation from thread i on atomic variable x
- SP^i: A sequence point from thread i
- M_x^i: A data load/data store/acquire/release/sequence point by thread i (on variable x)

We use the following notation for ordering dependencies between memory events:

- $M_x^i \xrightarrow{po} M_y^i$: M_x^i is program ordered before M_y^i
- $R_x^i \xrightarrow{sw} A_x^j$: A release operation on atomic variable x in thread i "synchronizes with" [7] an acquire operation on atomic variable x in thread j.

We reason about an ordering relation over all memory events, *persist memory order* (PMO), denoted as $\leq_p$. An ordering relation between stores in PMO implies the corresponding persist actions are ordered; that is,

$A \leq_p B \rightarrow B$ may not persist before A.

Memory events can be ordered in PMO using a combination of *intra-thread* and *inter-thread* ordering relations. Programmers can use the following guarantees to ensure a desired event order in PMO.

Ensuring intra-thread ordering: Based on the ordering guarantees provided by the language (via sequence points 3.2.4) intra-thread ordering can be achieved as follows:

Sequence point guarantee: If two memory events on the same thread are separated by a sequence point in program order, then they are ordered in PMO. Formally:

$$M_x^i \xrightarrow{po} SP^i \xrightarrow{po} M_y^i \rightarrow M_x^i \leq_p M_y^i \quad (1)$$

Note that we use the existing `std::atomic_thread_fence` instruction in C++11 as our sequence points.

Ensuring inter-thread ordering: Inter-thread ordering is achieved using the "synchronizes with" [7] relationship between a release and a subsequent acquire operation.

Synchronization guarantee: If two memory events are ordered via synchronization accesses, then they are ordered in PMO. Formally:

$$M_x^i \xrightarrow{po} R_s^i \xrightarrow{sw} A_s^j \xrightarrow{po} M_y^j \rightarrow M_x^i \leq_p M_y^j \quad (2)$$

Furthermore, PMO is a *transitive* (and irreflexive) ordering relationship, that is:

Transitivity guarantee: If A is ordered before B in PMO and B is ordered before C in PMO, then A is ordered before C in PMO. Formally:

$$M_x^i \leq_p M_y^j \wedge M_y^j \leq_p M_z^k \rightarrow M_x^i \leq_p M_z^k \quad (3)$$

A programmer can use the above three guarantees to express the desired order of persists. It is the responsibility of the compiler to translate these constraints to machine code using the ISA-level persistency model, and it is the responsibility of the hardware to enforce these constraints. Enforcing constraints on persists is expensive (due to the high access latencies of PMs), so, it is important to co-design language-level persistency models, ISA-level persistency models, and hardware implementations such that only the necessary constraints are enforced.

4.2 Mapping to ISA-level persistency

While ARP can be translated to any epoch-based ISA-level persistency model [14, 21, 25, 30, 41], in this paper, we provide mappings to the state-of-the-art RCBSP model [30]. One of the advantages of RCBSP is that it is a *strict persistency* model; that is, if the compiler

ARP memory events	RCBSP mapping	Ideal mapping
Data load/store on addr a	`ldr/str a;`	`ld/st a;`
Seq. Pt. (SP)	`dmb ish;`	`full;`
Store Release on addr a	`dmb ish; str a;`	`rel a;`
Load Acquire on addr a	`ldr a; dmb ish;`	`acq a;`

Table 1: Mapping from ARP memory events to RCBSP [30], which is based on ARMv7a. Ideal mappings from ARP are for an ISA that supports release consistency.

Benchmark	Fence directionality	Volatile annotations
cq	1.4×	2.1×
pc	1.9×	5.7×
sps	1.7×	2.9×
TATP	2.7×	3.6×
TPCC	1.6×	13.1×
YCSB_A	1.8×	5.9×

Table 2: Increase in persists per epoch when the memory controller is aware of fence directionality and volatile fences.

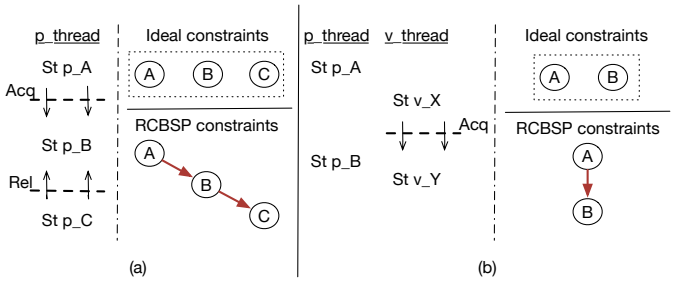

Figure 2: (a) Unnecessary constraints enforced due to hardware being oblivious to fence directions. (b) Unnecessary constraints enforced due to lack of language-level semantics to express volatile fences.

ensures that two stores are ordered by the ISA-level consistency model, then the corresponding persists are ordered as well.

Table 1 lists four important kinds of memory events in ARP and how they map to the machine ISA under RCBSP. Non-synchronization data accesses translate to regular loads and stores. A sequence point is translated to a full fence instruction (`DMB ISH` in ARM). A store release operation is translated to a full fence followed by a regular store instruction. A load acquire operation is translated to a regular load followed by a full fence instruction.

4.3 Fence directionality

We next discuss two sources of unnecessary persist constraints that arise when mapping ARP to RCBSP. The first arises because of the differences between the underlying consistency models of ARP and RCBSP. While ARP (and the C++11 memory model) is based on release consistency [19], RCBSP is based on the more conservative ARMv7 consistency model. Hence, RCBSP is oblivious to uni-directional acquire and release operations that are available in C++11 and ISAs based on release consistency (e.g., ARMv8).

ARP allows programmers to use uni-directional synchronization operations (*acq* and *rel*) to order memory accesses. Both *acq* and *rel* operations are usually used to ensure memory accesses within a critical section do not "leak out", however, they allow memory accesses from outside the critical section to "leak into" the critical section. However, as ARMv7 does not distinguish between an *acq* and a *rel*, compilers are forced to use a full fence (`DMB ISH` [48], which precludes memory access reordering in both directions) for both of them. Figure 2 (a) shows the unnecessary ordering constraints caused by using a full fence instead of uni-directional *acq* or *rel*. A thread performs stores to three persistent addresses, A, B, and C. The stores to A and B are separated by an *acq*, while stores to B and C are separated by a *rel*. As per the semantics of ARP, all three of A, B and C are considered concurrent and may execute and persist in any order. However, replacing the *acq* and *rel* with a full fence requires that persists to A, B, and C are serialized. So, persist order is overconstrained by RCBSP. Table 2 shows the increase in persists per epoch possible by distinguishing the required directionality of a fence. Such over-constraints on persist order are not specific to RCBSP, but arise whenever the ISA-level persistency model is stricter than the language-level persistency model.

Table 1 also shows the mapping of the four C++11 memory events to an ISA that provides uni-directional acquire and release operations (e.g., ARMv8). A store release is translated to a corresponding release instruction and a load acquire to an acquire instruction.

4.4 Conflating sync. with recoverability

The second set of unnecessary constraints are caused by the lack of mechanisms to allow programmers to annotate constraints that are required for concurrency control, but not for recoverability. Consider the case in Figure 2 (b), where two unrelated threads (*p_thread* and *v_thread*) issue memory accesses. RCBSP serializes persists and fences from all cores into the write queue at the PM controller. So, if the *acq* from *v_thread* happens to arrive at the PM controller between the two persists requests from *p_thread*, then the PM controller will place them in different epochs, introducing an unnecessary constraint.

Ideally, we would like the hardware to enforce only constraints required for recovery, however, accurately tracking these persist constraints over multiple cores is challenging. Instead, we observe that programmers can identify *acq* and *rel* memory operations that have no persist semantics (i.e., they are required for concurrency control but were never meant to order persists). For example, some threads may never issue any persist operations and communicate only among themselves [59]. With minor extensions to the C++11 memory model, programmers can annotate *acq* and *rel* that do not have persist semantics as non-persistent or "volatile". And, with appropriate extensions to the machine ISA, this information can be passed to hardware, avoiding unneeded persist constraints to improve performance. Table 2 shows the increase in persists per epoch possible by making sure that volatile *acq* and *rel* are not sent to the PM controller.

Discussion: Mitigating the two sources of unnecessary persist constraints allows more persists to join each epoch. Larger epochs in turn provide the PM controller greater flexibility to schedule and batch persist operations, improving persist concurrency, leading to substantial performance gains since PM write latencies are so high.

Algorithm 1 Epoch allocation in PM controller

```
Input: type of barrier barrierType, persists S, wait for acquire flag waitForAcq
1: if waitForAcq && barrierType == p_acq then
2:     epoch = epoch + 1
3:     waitForAcq.reset()
4: else if barrierType == p_rel then
5:     waitForAcq.set()
6: else if barrierType == p_full then
7:     epoch = epoch + 1
8: else
9:     S.epoch = epoch
10: end if
```

5 EXTENDING RCBSP FOR ARP

We extend RCBSP [30] to support ARP with unidirectional and volatile fences. The key change to the RCBSP hardware is to allow a single persist buffer entry to represent both a persist and a fence for store-release and store-fence operations. However, the PM controller's epoch-based scheduling mechanism must be redesigned to account for the fence-directionality that ARP provides.

5.1 Enforcing unidirectional fences

In RCBSP, the PM controller tracks persists in epochs and maintains a current epoch number, to which newly arriving persists are assigned. The PM controller drains persists in epoch order. Since RCBSP only supported full fences, the PM controller increments the current epoch number upon each fence. We extend RCBSP to support ARP's unidirectional fences by changing the algorithm for incrementing the current epoch number. The current epoch number is incremented only: (1) upon receiving a full fence, or (2) upon receiving the first *acq* after a *rel*. A full fence always creates a new epoch, since it orders all persists that precede/follow it. However, the respective directionalities of an *acq* and a *rel* mean that only a (*rel* + *acq*) combination disallows persists prior to the *rel* from reordering with persists after the *acq*. Successive *acq*s and *rel*s do not impact the current epoch number. Algorithm 1 provides pseudo-code for the epoch management algorithm, which uses the waitForAcq flag to indicate whether the next acquire operation should open a new epoch. We illustrate the assignment of epochs for the following scenarios:

Conflicting acquire-release blocks: Figure 3 (a) illustrates two threads updating a conflicting address `P_X` in persistent memory. Core-0 acquires a lock that resides in volatile memory V_A_0 (L_A^0), sets the persistent location `P_X` (S_X^0), and then releases the lock (S_A^0). It then sets a persistent location `P_Y` (S_Y^0) after releasing the lock. Core 1 then proceeds to acquire lock V_A_0 (L_A^1) and updates location `P_X` (S_X^1). It then releases the lock and sets persistent location `P_Z` (S_Z^1). Assume that the current epoch number is 0 at the start of this code sequence and is incremented to 1 upon the first *acq*.

The dependency tracking mechanism at the persist buffers preserves the happens-before ordering between the release of lock S_A^0 by core-0 and acquire of lock L_A^1 by core-1, and drains the stores to the PM controller in the order shown in Figure 3 (a). At the PM controller, upon receiving the lock release by core-0 *rel* S_A^0, the PM sets waitForAcq, indicating that the next acquire must initiate a new epoch. The next persist, S_Y^0, is still assigned to ongoing epoch 1. Upon receiving *p_acq* L_A^1 by core-1, because waitForAcq is set, the current epoch is incremented to 2. Subsequent persists S_X^1 and S_Z^1 arrive and are assigned to epoch 2. Note that persists lying between a release and subsequent acquire may join either epoch. To minimize re-ordering complexity, we assign these persists to the prior epoch. The persists in epoch 2, S_X^1 and S_Z^1, cannot be re-ordered with the persists in epoch 1, S_X^0 and S_Y^0. As a result, the shared address `X` is updated in the persistent memory in the order the stores were executed.

Interleaved acquire-release blocks: The example in Figure 3 (b) depicts two threads accessing separate regions of persistent memory by acquiring distinct locks. As in the previous example, the PM controller increments the epoch number to 1 upon receiving *acq* L_{A0}^0 and resets the waitForAcq flag. As core 1 then acquires a different lock, *acq* L_{A1}^1 has no dependency in the persist buffer and drains immediately to the PM controller. Since there has been no release since the last acquire (waitForAcq is clear), *acq* L_{A1}^1 does not increment the epoch number. Upon receiving *rel* S_{A1}^1 from core 1, the waitForAcq flag is set. The subsequent release operation *rel* S_{A0}^0 has no effect; the arriving persist S_Y^0 is assigned to epoch 1. Note that the persists within both critical sections are concurrent and join the same epoch.

5.2 Extensions for volatile annotations

To allow programmers to annotate *acq* and *rel* as being "volatile-only", we propose to add an argument to C++11 sync (`std::atomic`) variable accesses. In addition to the memory order argument for atomic accesses (`std::memory_order`), we introduce a new argument that identifies if the memory access has persistent semantics (`bool is_persistent`). By default, sync accesses are labeled as persistent (`is_persistent = true`). For instance, the new definition of a load on an atomic variable (x), is then:

```
x.load(std::memory_order,
                    bool is_persistent = true);
```

This new load operation is synchronized with other variables in the program as per the specific memory order, however, the newly introduced `is_persistent` flag is used to inform the hardware whether the load operation is intended to have any impact on the order of persists. Similarly, in the machine ISA, we add "volatile" (non-persistent) versions of *acq*, *rel*, and fence instructions, allowing the compiler to map persistent/volatile sync accesses to the ISA. The persistent and volatile variants of *acq*, *rel*, and fence have the same behavior, except that the volatile versions are not sent to persist buffers and have no effect on subsequent persists.

6 EVALUATION

We study the relative performance of five different persistency models: (a) SCP (from section 3.2.3), (b) ISA-level RCBSP, (c) our hardware design for ARP, (d) our hardware design for ARP with volatile annotations (ARP+VA), and (e) an idealized performance limit model (Ideal). Under the ideal case, we artificially maintain a constant 64 (size of write queue) persists per epoch to estimate an upper bound on performance. Note that, under Ideal, data structures are not recoverable in the event of failure; we include it only as a limit study.

Configuration: We model persistent memory using the timing model from Xu *et al.* [56] to represent phase-change memory operating at 533MHz with a 1KB row buffer. We model a persistent memory controller with a 64-entry write queue and schedule persists

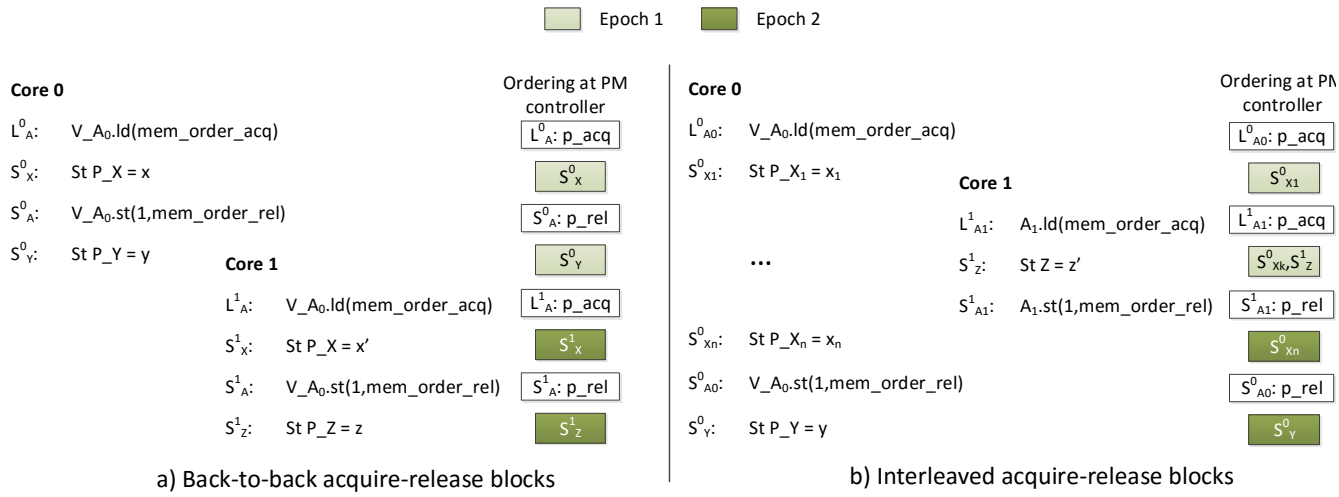

Figure 3: Allocation of epochs for unidirectional fences in the PM controller.

Core	8-cores, 2GHz OoO 6-wide Dispatch, 8-wide Commit 40-entry ROB 16/16-entry Load/Store Queue
I-Cache	32kB, 4-way, 64B 1ns cycle hit latency, 2 MSHRs
D-Cache	64kB, 8-way, 64B 2ns hit latency, 6 MSHRs
L2-Cache	8MB, 16-way, 64B 16ns hit latency, 16 MSHRs
Memory controller (DRAM, PM)	64/32-entry write/read queue, 1kB row buffer
DRAM	DDR3, 800MHz
PCM	533MhZ, timing from [56]

Table 3: Simulator Specifications.

Benchmark	Description	PKC
Conc. queue	Insert/Delete entries in a queue	17.4
Persistent Cache	Persistent hash table	22.7
Array Swaps	Random swaps of array elements	41.8
TATP	Update location trans. in TATP [39]	30.8
TPCC	New Order trans. in TPCC [49]	11.7
YCSB_A	YCSB Workload A [15]	17.4

Table 4: Benchmarks. PKC = persists per 1000 cycles

using an FR-FCFS policy [46], subject to persist ordering constraints. We extend our compiler's `std::atomic` implementation to support our C++11 extensions with volatile annotations in the ARP+VA model. Table 3 provides a summary of our system configuration.

Benchmarks: We study a suite of three PM-centric multithreaded micro-benchmarks, described in Table 4. Our Concurrent Queue (cq) is similar to that of Pelley [41], Array Swap (sps) is similar to that in NV-Heaps [13], and Persistent Cache (pc) is a persistent hash table similar to [25]. In addition, we also consider three write-intensive benchmarks. TATP [39] and TPCC [39] execute "update location" and "new order" transactions, respectively, on top of a transactional storage manger designed for persistent memory, similar to [29]. YCSB `A` [15] (YCSB_A) is a write intensive key-value store workload with 50% reads and 50% updates. It runs on a custom key-value store that has been designed to support all five of our persistency models.

We select these benchmarks specifically because of their PM write-intensiveness. As a measure of "write-intensive"-ness, we report the number of persists issued per 1000 cycles (PKC) in Table 4. Array swap is our most write-intensive micro-benchmark while concurrent queue is the least, so we expect them to show the most and least sensitivity to different persistency models. Similarly, TATP and TPCC are respectively the most and least write-intensive benchmarks.

All workloads run with eight worker threads that update the underlying persistent data-structure. In all the benchmarks, we run an additional work allocator thread [50] and two volatile antagonist threads to evaluate the proposed volatile annotations. Each worker thread has a 64-entry work queue that resides in the volatile memory. The work allocator thread distributes tasks from a shared work queue to the eight worker threads. Since the work queue resides in volatile memory, the acquire and release fences required to order work queue accesses are volatile fences. This work queue structure represents the request dispatch of a typical network application and illustrates how threads that issue no accesses to persistent memory can nevertheless impact persist performance indirectly due to synchronization operations. Each workload also includes two antagonist threads to simulate the traffic of background threads polling for events. The two threads contend on a lock to a shared counter in volatile memory, increment it, and release the lock. These antagonists represent synchronization activity by unrelated application threads and do not interact directly with the eight worker threads.

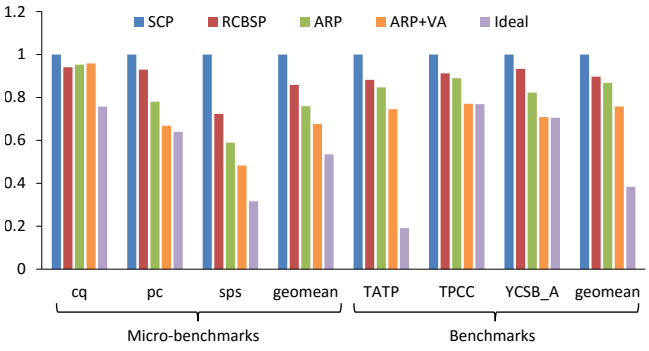

Figure 4: Execution time normalized to SCP: The graph compares execution time of ARP and ARP+VA with SCP and RCBSP for micro-benchmarks and benchmarks.

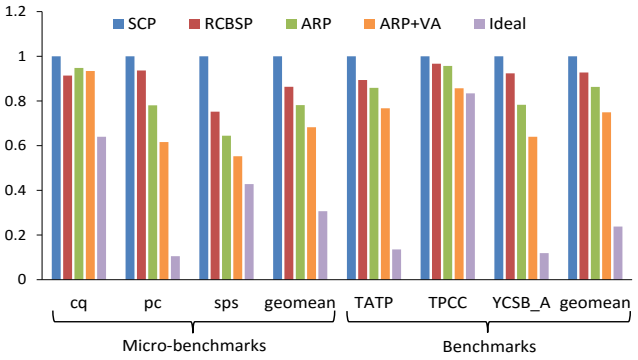

Figure 5: Page miss rate normalized to SCP: Lower page miss rate in the PM controller implies better persist scheduling.

Benchmark	SCP	RCBSP	ARP	ARP+VA
cq	1	1.7	2.3	3.5
pc	1	2.2	3.9	12.3
sps	1	4.6	8.1	13.2
TATP	1	1.7	4.5	6.0
TPCC	1	1.7	2.7	22.1
YCSB_A	1	2.1	3.8	12.5

Table 5: Persists per epoch

6.1 Performance comparison

We first present the opportunities ARP and ARP+VA have to improve performance and then show the realized gains.

Persists per epoch: Table 5 shows the persists per epoch under each persistency model. More persists per epoch allow greater reordering opportunity and better persist scheduling at the PM controller. ARP exploits unidirectional acquire and release operations to reduce the number of epochs at the PM controller and increase persists per epoch. ARP provides a 3.9× and 1.8× increase in persists per epoch relative to SCP (which by definition places each persist in its own epoch) and RCBSP, respectively. Further, ARP+VA distinguishes volatile and persistent fences using programmer inserted volatile annotations and achieves a 9.9× and 4.6× increase in persists per epoch relative to SCP and RCBSP.

Micro-benchmarks: The left set of bars in Figure 4 contrast the execution time for micro-benchmarks under RCBSP, ARP, ARP+VA, and Ideal ordering models normalized to SCP. Array swap (sps) gains the most from ARP+VA with 51.7% performance improvement over SCP and 33.2% over RCBSP. As evident from the ideal result, array swap is sensitive to the increase in persists per epoch. Concurrent queue (cq) gains the least. In this microbenchmark, entries are pushed or popped from the queue serially by the worker threads; there is limited thread concurrency. As a result, it is not sensitive to the number of persists per epoch and gains little performance even under the ideal case. In fact, due to inopportune read-write bus turnarounds, performance with ARP+VA slightly degrades relative to RCBSP. Overall, ARP+VA improves micro-benchmark execution time by 32.4% as compared to SCP and 21.2% as compared to RCBSP.

Benchmarks: Figure 4 also contrasts the execution time of the TATP, TPCC, and YCSB_A benchmarks under each persistency model. YCSB_A is the most sensitive, gaining 17.8% and 29.2% performance, respectively, under ARP and ARP+VA. Further, ARP+VA improves execution time of TATP by 25.5%, and TPCC by 23% as compared to SCP. It is interesting to note that unidirectional fences in ARP do not provide substantial performance gain over RCBSP in TATP even though the ideal case outperforms SCP by 70.2%. TATP includes numerous small critical sections containing full fences to log values before updating the persistent database, limiting potential performance gains. The majority of the gain for TATP is achieved by annotating volatile fences explicitly. Overall, ARP+VA improves execution time of the three benchmarks by 24.3% and 15.5% over SCP and RCBSP respectively.

6.2 Persist scheduling

Finally, we report the impact of PM scheduling policies on the page miss rate of the PM controller in Figure 5. The PM controller's FR-FCFS policy seeks to maximize page hits within each persistent memory bank. As described earlier, increasing the number of persists per epoch improves the scheduling flexibility available to the controller. Owing to the unidirectional fences in ARP, the page miss rate drops on average by 17.9% relative to SCP and 8.2% relative to RCBSP. This improvement is the result of the increase in the number of persists that can be scheduled to write to different PM banks concurrently. ARP+VA further relaxes persist ordering constraints by distinguishing volatile and persistent fences, achieving a further 13.0% improvement in page miss rate relative to ARP. The ideal model lowers page miss rate by 76.3% over SCP, indicating the upper bound on PM bank-level parallelism available in these workloads. However, it should be noted that this model does not maintain ordering between the persists and data structures are not recoverable in the event of failure.

7 RELATED WORK

We discuss related work on future system support for PM.

Software-based persistency solutions: A variety of works offer interfaces for programming persistent data structures. NV-Heaps [13] and Mnemosyne [53] offer abstractions for building persistent objects using a set of primitives like memory allocation and atomic

sections. REWIND [11] proposes a user-mode library that can directly perform transactional updates to underlying persistent data structures. SCMFS [55] and Aerie [52] propose file systems for storage-class memory. Atlas [10] provides durability semantics for lock-based code. It employs undo-logging to provide atomicity of updates in FASE, the failure atomic critical sections. Our work adds persist semantics to the language memory model and facilitates portability of these software proposals.

Hardware-based persistency solutions: Several works extend ISAs with epoch-based abstractions and epoch barriers to order persists. Pelley et al. [41] proposes memory persistency models closely associated with hardware consistency models that allow programmers to describe persist order constraints. BPFS [14] uses epoch barriers to divide program execution into different epochs; stores within an epoch are concurrent while those in different epochs persist in order. Joshi et al. [25] proposes efficient persist barriers that minimize the number of cache line write-backs on the critical path. HOPS [36], like DPO [30], extends the cache hierarchy to separately enforce the ordering constraints between persists from their respective durability constraints. Izraelevitz et al. [24] provides an automatic transformation technique for non-blocking data structures to convert them to recoverable data structures under various persistency models. Our work is instead focused on exploring the design space for persistent memory programming primitives at the language level and designing hardware to implement these primitives. Bhandari et al. [5] show that write-through caching sometimes provides better performance than write-back caching. Kiln [58], LOC [33], and ATOM [26] provide a storage transaction interface (providing atomicity, consistency and durability) to PM, wherein the programmer must ensure isolation. Our work builds on delegated ordering [30], which seeks to increase overlap between program execution and persist operations.

Ordering at the PM controller: Doshi et al. [17] propose a non-intrusive PM controller that provides atomicity of persistent memory transactions. It builds a victim cache to hold the evictions from caches and a redo-logging mechanism to provide atomicity. FIRM [59] proposes persist-aware memory scheduling to improve bank-level parallelism for persistent updates. Other works, such as [32, 44, 60], optimize resource allocation at PM controller for application fairness while maintaining persist order guarantees. Our work may compose with their proposals, exposing greater scheduling freedom from the language memory model to the memory controller.

Periodic checkpoint-based solutions: ThyNVM [45] proposes a checkpointing mechanism to provide crash consistency support. ThyNVM uses dynamically adaptable checkpoint granularity and pipelining to improve checkpointing efficiency. Survive [35] is a novel DRAM architecture for efficient checkpointing in systems with PM.

PM write endurance and latency: Other works focus on other aspects of PM devices, such as scalability or energy savings. Some works [31, 61] seek to replace DRAMs with PCM-based memories to improve capacity scalability while addressing challenges like write endurance and longer write latency. Other works, such as [42, 43], address write endurance challenges through wear-leveling techniques, while [12, 20, 56, 57] accelerate writes to the PM. These important concerns are orthogonal to our focus.

Other: A group of studies considers persistent memory in the context of systems with persistent caches, i.e., stores become durable as soon as they execute. Cache persistence can be achieved using non-volatile devices [54, 58], by ensuring that a battery backup is available to flush the contents of caches to PM upon power failure [37, 38], or by not caching PM accesses [54]. Integrating non-volatile devices and high-perfor-mance logic poses manufacturability challenges. It is unclear if mechanisms that flush the cache upon power failure are viable for systems with large caches. Our work assumes volatile cache hierarchies.

8 CONCLUSION

Past work requires programmers to reason about ISA-level persistency models to develop recoverable software for persistent memory systems, hurting software portability. We examined extending the language-level memory model to provide ordering guarantees on persists. We presented a taxonomy of differing guarantees that a language-level persistency model might provide and proposed acquire-release persistency, a language level persistency model for C++11. We then co-optimized ARP with an underlying ISA-level persistency model, RCBSP, to minimize the number of persist constraints the PM controller must enforce, substantially increasing PM bank-level parallelism and performance.

9 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by ARM and the National Science Foundation under the award NSF-CCF-1525372.

REFERENCES

[1] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *Computer* 29, 12 (Dec. 1996), 66–76. https://doi.org/10.1109/2.546611

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. 2005. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*. IEEE Computer Society, Washington, DC, USA, 316–327. https://doi.org/10.1109/HPCA.2005.41

[3] ARM. 2016. ARMv8-A architecture evolution. (2016). https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution.

[4] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 416–427. http://dl.acm.org/citation.cfm?id=2337159.2337207

[5] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2012. *Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming*. Technical Report HPL-2012-236. Hewlett-Packard.

[6] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. 2009. InvisiFence: Performance-transparent Memory Ordering in Conventional Multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1555754.1555785

[7] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

[8] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 55–67. https://doi.org/10.1145/2926697.2926704

[9] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 278–289. https://doi.org/10.1145/1250662.1250697

[10] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[11] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. *PVLDB* 8, 5 (2015), 497–508. http://www.vldb.org/pvldb/vol8/p497-chatzistergiou.pdf

[12] Sangyeun Cho and Hyunjin Lee. 2009. Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 347–357. https://doi.org/10.1145/1669112.1669157

[13] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[14] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[16] Timothy J Dell. 1997. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics Division* (1997), 1–23.

[17] Kshitij Doshi, Ellis Giles, and Peter J. Varman. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*.

[18] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[19] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/325164.325102

[20] Andrew Hay, Karin Strauss, Timothy Sherwood, Gabriel H. Loh, and Doug Burger. 2011. Preventing PCM Banks from Seizing Too Much Power. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 186–195. https://doi.org/10.1145/2155620.2155642

[21] Intel. 2014. Intel Architecture Instruction Set Extensions Programming Reference (319433-022). (2014). https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[22] Intel and Micron. 2015. Intel and Micron Produce Breakthrough Memory Technology. (2015). http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[23] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[24] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing: 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, Cyril Gavoille and David Ilcinkas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[25] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2830772.2830805

[26] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Support for Logging. In *23rd International Conference on High-Performance Computer Architecture (HPCA-23 2017)*. 1–12.

[27] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*. 1–12. https://doi.org/10.1109/HPCA.2010.5416658

[28] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, Washington, DC, USA, 65–76. https://doi.org/10.1109/MICRO.2010.51

[29] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[30] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated persist ordering. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. 1–13. https://doi.org/10.1109/MICRO.2016.7783761

[31] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[32] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. 2014. NVM Duet: Unified Working Memory and Persistent Store Architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 455–470. https://doi.org/10.1145/2541940.2541957

[33] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19-22, 2014*. 216–223. https://doi.org/10.1109/ICCD.2014.6974684

[34] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. 2010. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 210–221. https://doi.org/10.1145/1815961.1815987

[35] Amirhossein Mirhosseini, Aditya Agrawal, and Josep Torrellas. 2016. Survive: Pointer-based In-DRAM Incremental Checkpointing for Low-Cost Data Persistence and Rollback-Recovery. *IEEE Computer Architecture Letters* (2016).

[36] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*.

[37] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/2150976.2151018

[38] Faisal Nawab, Dhruva Chakrabarti, Terence Kelly, and Charles B. Morey III. 2014. *Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience*. Technical Report HPL-2014-70. Hewlett-Packard.

[39] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. 2011. Telecom Application Transaction Processing Benchmark. (2011). http://tatpbenchmark.sourceforge.net/.

[40] Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2013. . . . And Region Serializability for All. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*.

[41] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[42] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 14–23. https://doi.org/10.1145/1669112.1669117

[43] Moinuddin K. Qureshi, Andre Seznec, Luis A. Lastras, and Michele M. Franceschini. 2011. Practical and Secure PCM Systems by Online Detection of Malicious Write Streams. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 478–489. http://dl.acm.org/citation.cfm?id=2014698.2014882

[44] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 24–33. https:

//doi.org/10.1145/1555754.1555760

[45] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 672–685. https://doi.org/10.1145/2830772.2830802

[46] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA '00)*. ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/339647.339668

[47] Aritra Sengupta, Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Milind Kulkarni. 2015. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 561–575. https://doi.org/10.1145/2786763.2694379

[48] Jaroslav Sevcik and Peter Sewell. 2011. C/C++11 mappings to processors. (2011). https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

[49] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark B. (2010). http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf.

[50] Ten H Tzen and Lionel M Ni. 1991. Dynamic Loop Scheduling for Share-Memory Multiprocessors.. In *ICPP (2)*. 247–250.

[51] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1960475.1960480

[52] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 14, 14 pages. https://doi.org/10.1145/2592798.2592810

[53] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[54] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging Through Emerging Non-volatile Memory. *Proc. VLDB Endow.* 7, 10 (June 2014), 865–876. https://doi.org/10.14778/2732951.2732960

[55] Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2063384.2063436

[56] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie. 2015. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 476–488. https://doi.org/10.1109/HPCA.2015.7056056

[57] Jianhui Yue and Yifeng Zhu. 2013. Accelerating Write by Exploiting PCM Asymmetries. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 282–293. https://doi.org/10.1109/HPCA.2013.6522326

[58] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744

[59] Jishen Zhao, Onur Mutlu, and Yuan Xie. 2014. FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 153–165. https://doi.org/10.1109/MICRO.2014.47

[60] P. Zhou, Y. Du, Y. Zhang, and J. Yang. 2010. Fine-grained QoS scheduling for PCM-based main memory systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–12. https://doi.org/10.1109/IPDPS.2010.5470451

[61] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM, New York, NY, USA, 14–23. https://doi.org/10.1145/1555754.1555759