Weak Persistency Semantics from the Ground Up

Formalising the Persistency Semantics of ARMv8 and Transactional Models

AZALEA RAAD, MPI-SWS, Germany
JOHN WICKERSON, Imperial College London, UK
VIKTOR VAFEIADIS, MPI-SWS, Germany

Emerging non-volatile memory (NVM) technologies promise the durability of disks with the performance of volatile memory (RAM). To describe the persistency guarantees of NVM, several memory persistency models have been proposed in the literature. However, the formal persistency semantics of mainstream hardware is unexplored to date. To close this gap, we present a formal declarative framework for describing concurrency models in the NVM context, and then develop the PARMv8 persistency model as an instance of our framework, formalising the persistency semantics of the ARMv8 architecture for the first time. To facilitate correct persistent programming, we study transactions as a simple abstraction for concurrency and persistency control. We thus develop the PSER (persistent serialisability) persistency model, formalising transactional semantics in the NVM context for the first time, and demonstrate that PSER correctly compiles to PARMv8. This then enables programmers to write correct, concurrent and persistent programs, without having to understand the low-level architecture-specific persistency semantics of the underlying hardware.

CCS Concepts: • **Theory of computation → Concurrency**; **Semantics and reasoning**; • **Software and its engineering → General programming languages**.

Additional Key Words and Phrases: weak memory, memory persistency, non-volatile memory, ARMv8

ACM Reference Format:
Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019. Weak Persistency Semantics from the Ground Up: Formalising the Persistency Semantics of ARMv8 and Transactional Models. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 135 (October 2019), 27 pages. https://doi.org/10.1145/3360561

1 INTRODUCTION

Computer storage is traditionally divided into two categories: fast, *volatile*, byte-addressable memory (e.g. DRAM), which loses its contents in case of a power failure, and slow, *persistent*, block-addressable storage (e.g. hard drives), which preserves its contents in case of a power failure. Due to this split, applications typically maintain their data structures in memory and periodically write important data to disk. However, emerging new technologies in *non-volatile memory* (NVM) [Kawahara et al. 2012; Lee et al. 2009; Strukov et al. 2008] may soon render this dichotomy obsolete by enabling processors to access data guaranteed to persist a power failure at byte-level granularity and at performance comparable to regular (volatile) RAM. It is widely believed that NVM (a.k.a. persistent memory) will eventually supplant volatile memory, allowing for efficient access to persistent data [Intel 2014; ITRS 2011; Pelley et al. 2014]. As such, the NVM literature has grown rapidly over the recent years [Boehm and Chakrabarti 2016; Chakrabarti et al. 2014; Chatzistergiou

Authors' addresses: Azalea Raad, MPI-SWS, Saarland Informatics Campus, Germany, azalea@mpi-sws.org; John Wickerson, Imperial College London, 180 Queen's Gate, London, UK, j.wickerson@imperial.ac.uk; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.

2475-1421/2019/10-ART135
https://doi.org/10.1145/3360561

et al. 2015; Coburn et al. 2011; Gogte et al. 2018; Izraelevitz et al. 2016a; Kolli et al. 2017, 2016a,b; Nawab et al. 2017; Raad and Vafeiadis 2018; Volos et al. 2011; Wu and Reddy 2011; Zhao et al. 2013].

Using persistent memory *correctly*, however, is not easy. A key challenge is ensuring correct recovery after a crash (e.g. a power failure) by maintaining the consistency of data in memory, which requires an understanding of the order in which writes are propagated to memory. The problem is that CPUs are not directly connected to memory; instead there are multiple volatile caches in between. As such, writes may not propagate to memory at the time and in the order that the processor issues them, but rather at a later time and in the order decided by the cache coherence protocol. This can lead to surprising outcomes. For instance, consider the simple sequential program $x := 1; y := 1$, running to completion and crashing thereafter. On restarting the computer, the memory may contain y=1, x=0; i.e. the $x := 1$ write may not have propagated to memory before the crash.

To ensure correct recovery, one must thus control the order in which writes are propagated to persistent memory. To this end, Pelley et al. [2014] introduced the notion of *persistency models* to formally define the persistency semantics of programs (the permitted behaviours upon recovery) by prescribing the order in which writes are persisted to memory. Existing literature includes several proposals of persistency models varying in strength and performance [Condit et al. 2009; Gogte et al. 2018; Izraelevitz et al. 2016b; Joshi et al. 2015; Kolli et al. 2017, 2016b; Raad and Vafeiadis 2018]. However, to our knowledge, the persistency semantics of *mainstream* hardware such as the ARM [ARM 2018] architecture remains unexplored to date.

To address this, we formalise the persistency semantics of the ARM architecture for the first time to our knowledge. To define our declarative semantics, we develop a general framework for describing declarative concurrency models in the context of persistent memory. We then develop the PARMv8 (persistent ARMv8) model as an extension of the ARMv8 (weak) memory model by Pulte et al. [2018], and present PARMv8 as an instance of our general framework.

Although it is crucial to support the nascent NVM technologies at the hardware level, programmers rarely develop code on top of low-level hardware models such as PARMv8. To support persistent programming in high-level languages such as C/C++, researchers have proposed *language-level persistency* models instead [Gogte et al. 2018; Kolli et al. 2017], and the C++ committee has called for a study group to incorporate NVM support into C++ in the near future [Douglas 2018]. Whilst no mainstream language currently supports persistent programming, existing language-level persistency models remain too *low-level*. In particular, existing work does not provide high-level concurrency control mechanisms (e.g. transactions) readily available to programmers in traditional volatile settings, making correct persistent programming inaccessible to the uninitiated programmer. This has led Intel to develop PMDK [Intel 2015], providing a *transactional* persistence library.

To formalise the semantics of transactions in the presence of NVM, we develop the PSER (persistent serialisability) model as an extension of the well-known transactional consistency model: serialisability. To our knowledge, PSER is the first formal transactional consistency and persistency model in the NVM context. To show that PSER is useful, we demonstrate that PSER can be used to convert *any* correct sequential implementation of a library to a *correct, concurrent* and *persistent* implementation of the same library. Moreover, to show that PSER is feasible, we develop a *sound* implementation of PSER in PARMv8, thereby showing that PSER *correctly compiles* to PARMv8.

Related Work. Although the existing literature on non-volatile memory has grown rapidly in the recent years, *formalising* persistency models has largely remained unexplored to date.

At the hardware level, existing literature includes several persistency models. Pelley et al. [2014] describe several such models, including *epoch persistency*, under *sequentially consistent* (SC) machines, whilst Condit et al. [2009]; Joshi et al. [2015] describe epoch persistency under *'total-store-order'* (TSO) machines. However, neither work provides a *formal* description of the studied

persistency semantics (declarative or operational). Izraelevitz et al. [2016b] give a formal declarative semantics for epoch persistency under the release consistency model [Gharachorloo et al. 1990] using abstract executions, whilst Raad and Vafeiadis [2018] formalise epoch persistency under the TSO model. However, neither work formalises the existing persistency semantics of *mainstream* hardware such as Intel-x86 [Intel 2019] and ARM [ARM 2018]. In particular, the work of Raad and Vafeiadis [2018] proposes a *potential* persistency model for Intel-x86 which is rather different from the existing Intel-x86 model described informally in [Intel 2019]. Liu et al. [2019] develop the PMTest testing framework for finding persistency bugs in software running over hardware models. However, they do not formalise the persistency semantics of the underlying hardware. We believe that our formal PARMv8 model can provide a more rigorous foundation for tools such as PMTest.

At the software level, Kolli et al. [2017] propose *acquire-release persistency* (ARP), as an analogue to release-acquire consistency in C/C++. Gogte et al. [2018] propose the notion of *synchronisation free regions* (regions delimited by synchronisation operations or system calls), to ensure that the state observed after recovery is at a frontier of past synchronization operations on each thread. Both approaches enjoy good performance and can be efficiently used by seasoned persistent programmers. Nevertheless, their semantic models are rather low-level, rendering them too complex for the inexperienced developers. The NVM community has thus moved towards high-level transactional approaches [Avni et al. 2015; Kolli et al. 2016a; Shu et al. 2018; Tavakkol et al. 2018], most notably that of PMDK by Intel [2015], which offers a transactional persistence library.

Additional Material. The proofs of all theorems stated in the paper are given in full in the technical appendix available at http://plv.mpi-sws.org/pmem/. We also provide machine-readable versions of our persistency models in the Alloy modelling language [Jackson 2012].

Contributions and Outline. Our contributions (detailed in §2) are as follows: (1) in §3 we develop a formal declarative framework for describing concurrency models in the NVM context; (2) in §4 we develop PARMv8 as the first formal model of the ARM persistency semantics; (3) in §5 we develop PSER as the first formal transactional model in the NVM context, and show its utility for correct, concurrent and persistent library implementations; (4) in §6 we develop a sound implementation of PSER in PARMv8, demonstrating correct PSER-to-PARMv8 compilation. Finally, in §7 we discuss future work and conclude.

2 OVERVIEW

We proceed with a brief background (§2.1) and an overview of our contributions (§2.2–§2.4).

2.1 Persistency Semantics

Memory Consistency and Persistency. Memory *consistency* models describe the permitted behaviours of programs by ensuring that memory operations follow certain rules. These rules in effect describe a (volatile) *memory order* which constrains the visible order of memory accesses (reads and writes). That is, the volatile memory order defines the admissible visible memory states between threads, which in turn allows memory operations to be reordered, while preserving the intended program behaviour. For instance, under the sequential consistency (SC) model [Lamport 1979], the volatile memory order is given by the total execution order present under SC. The existing literature includes several consistency models, both at the hardware (architecture) [Pulte et al. 2018; Sewell et al. 2010] and software (programming language) levels [Batty et al. 2011; Lahav et al. 2017].

Analogously, memory *persistency* models describe the permitted behaviours of programs upon recovering from a crash (e.g. due to a power failure) by defining a *persistent memory order* [Pelley et al. 2014]. The persistent memory order constrains the order in which the effects of instructions are committed to persistent memory. As such, any pair of writes ordered by the persistent memory

order may not be observed out of that order upon crash recovery. To distinguish the volatile and persistent memory orders, memory *stores* are differentiated from memory *persists*: the former denotes the process of making the effects of an instruction (e.g. a write) visible to other processors, whilst the latter denotes the process of committing the effects of an instruction durably to non-volatile memory. As such, the volatile memory order constrains the order on stores, whilst the persistent memory order constrains the order on persists. As with consistency models, persistency models may be at the hardware or software level.

Existing literature includes several proposals for persistency models [Gogte et al. 2018; Izraelevitz et al. 2016b; Kolli et al. 2017; Pelley et al. 2014; Raad and Vafeiadis 2018]. Generally, persistency models are categorised along two axes: (1) strict versus relaxed; and (2) unbuffered versus buffered.

Strict and Relaxed Persistency. As with consistency models, persistency models may be strict or relaxed. Under strict persistency the volatile and persistent memory orders coincide. For instance, in the case of the sequential consistency (SC) model, this means that the execution order determines not only the volatile memory order, but also the order in which writes are persisted to memory. However, strict persistency may introduce unnecessary dependencies between persists, thus hindering performance needlessly. To remedy this, Pelley et al. [2014] propose *relaxed* persistency models, where the volatile and persistent memory orders are separated. The authors propose several such models including *epoch persistency*, studied by Raad and Vafeiadis [2018]. As we discuss shortly, the ARMv8 architecture follows a relaxed persistency model [ARM 2018], while the persistent serialisability (PSER) model (described below) follows a strict persistency model.

Unbuffered and Buffered Persistency. This dichotomy denotes whether persists occur *synchronously* or *asynchronously*. Under unbuffered persistency, persists occur synchronously: when executing a store instruction, its effects are immediately committed to persistent memory; i.e. execution is stalled by persists. In order to improve performance, *persist buffering* has been proposed to allow memory persists to occur asynchronously [Condit et al. 2009; Izraelevitz et al. 2016b; Joshi et al. 2015]. That is, memory persists are buffered in a queue of write-backs to persistent memory. This way, persists occur after their corresponding stores and as prescribed by the persistent memory order; however, execution may proceed ahead of persists. As such, after recovering from a crash, only a *prefix* of the persistent memory order may have successfully persisted. As we describe shortly, both the ARMv8 [ARM 2018] architecture and the PSER model follow a buffered persistency model. When it is necessary to control the write-back of buffered persists explicitly (e.g. before performing I/O), buffered models typically offer *synchronous* write-back instructions (with varying granularity) that wait until relevant pending persists have been drained from the persistent buffer and committed to persistent memory. For instance, the epoch persistency model provides a *sync* instruction which commits *all* pending writes, while ARM provides *per-location write-back* instructions which commit all pending writes on a *given cache line* (set of memory locations).

2.2 Formal Declarative Persistency Models

In §3 we describe a general framework for declarative concurrency models in the context of persistent memory. We then present our PARMv8 and PSER persistency models (described shortly) as instances of this general framework in §4 and §5, respectively.

In the literature of declarative concurrency models, the traces of a concurrent program P are typically represented as a set of *complete* executions that do not crash. However, in order to model the crashing behaviour of programs in the presence of persistent memory, one cannot simply consider complete executions only. Instead, we define an *execution chain* C as a sequence $G_1, \cdots, G_n$, comprising n *eras*. That is, the $G_1, \cdots, G_n$ chain models an execution of a program that crashes $n-1$ times, with each G_i describing an execution era between two adjacent crashes. Each

execution era G_i denotes the traces of shared memory accesses generated by the program in that era. As is standard in the literature of declarative concurrency models, each G_i comprises a set of memory events (the G_i nodes), and a number of relations on events (the G_i edges), describing e.g. the execution control flow via the 'program order' relation po. In order to capture the persistent memory orderings, we extend each execution G_i to include a '*non-volatile-order*' relation, nvo, defining the order in which writes are persisted to memory.

Furthermore, in order to ensure correct recovery upon a crash, each program P is associated with a *recovery mechanism* describing the code to be executed upon recovery from a crash. As such, we model a *persistent program* as a pair $\langle \mathsf{P}, \mathtt{rec} \rangle$, where P denotes the original program to be executed, and rec denotes its recovery mechanism. As such, given a chain $G_1, \cdots, G_n$ of a persistent program $\langle \mathsf{P}, \mathtt{rec} \rangle$, G_1 describes the execution of the original program P up to the very first crash; and for $2 \leq i \leq n$, each G_i describes the execution of rec in the i^{th} era, recovering from the $(i{-}1)^{\text{st}}$ crash.

Alloy Encoding. When expressed in the declarative style, memory *consistency* models can be naturally encoded in the Alloy language [Jackson 2012], where they can be explored and compared [Wickerson et al. 2017], or used as a basis for generating conformance tests [Lustig et al. 2017]. Our memory *persistency* models are also declarative, and are hence amenable to analysis with Alloy. We have encoded our PARMv8 and PSER models in Alloy, and provide our Alloy model files in our supplementary material, along with sanity checks necessary for exploring the models.

2.3 Architecture-Level Persistency: The PARMv8 Model

We develop the *PARMv8 memory model*, formalising the persistency semantics of the ARMv8 architecture as described informally in [ARM 2018]. We specify PARMv8 as an extension of the ARMv8 model by Pulte et al. [2018]. We proceed with a brief account of the ARMv8 and PARMv8 models. Later in §4 we describe the PARMv8 semantics formally.

The ARMv8 Model. The ARMv8 consistency model formalised by Pulte et al. [2018] is a relaxed model that allows for a number of weak behaviours (not present under sequential consistency), due in part to instruction *reordering*. In particular, under the ARMv8 model the instructions in each thread may be executed out of order. Consider the following programs:[1]

$$\begin{array}{l|l} x := 1; & \mathsf{b} := y; \\ y := 1; & \mathsf{c} := x; \end{array} \qquad \text{(P1)} \qquad\qquad \begin{array}{l|l} x := 1; & \mathsf{b} := y; \\ \mathbf{DMB}_{\mathtt{full}}; & \mathbf{DMB}_{\mathtt{full}} \\ y := 1; & \mathsf{c} := x; \end{array} \qquad \text{(P2)}$$

In the absence of additional orderings imposed by e.g. memory barriers, the instructions in each thread may be reordered. As such, the $x := 1$ and $y := 1$ writes in the left thread (as well as the $\mathsf{b} := y$ and $\mathsf{c} := x$ reads in the right thread) of (P1) may be reordered, allowing the right thread to observe $\mathsf{b}{=}1 \wedge \mathsf{c}{=}0$. By contrast, in (P2) the instructions in each thread are separated by a $\mathbf{DMB}_{\mathtt{full}}$ (a full 'data memory barrier') instruction, prohibiting their reordering. As such, the right thread cannot observe $\mathsf{b}{=}1 \wedge \mathsf{c}{=}0$ in (P2).

The PARMv8 Model. As mentioned earlier, the ARM architecture follows a relaxed, buffered persistency model. The buffered persistency of PARMv8 is reflected in the example of Fig. 1a, corresponding to the program in the left thread of (P1). Due to the buffered model of PARMv8, if a crash occurs during the execution of this program, at crash time either write may or may not have already persisted and thus $x, y \in \{0, 1\}$ upon recovery. In particular, as discussed above, ARMv8 allows for the two writes to be reordered, and thus in case of a crash it is possible to

[1] In all our examples we use x, y, $\cdots$ for (shared) memory locations and use a, b, $\cdots$ for thread-local registers.

$x := 1;$ $y := 1;$	$x := 1;$ $\mathbf{DMB}_{\mathtt{full}};$ $y := 1;$	$x := 1;$ $\mathbf{wb}\ x;$ $y := 1;$	$x := 1$ $\mathbf{wb}\ x;$ $\mathbf{DSB}_{\mathtt{full}};$ $y := 1;$	$x := 1;$ $\mathbf{wb}\ x;$ $\mathbf{DSB}_{\mathtt{full}};$ $y := 1;$	$\mathtt{a} := y;$ $\mathbf{DMB}_{\mathtt{full}};$ $\mathbf{if}\ (a)$ $\quad z := 1;$
(a)	(b)	(c)	(d)	(e)	
rec: $x, y \in \{0, 1\}$	rec: $x, y \in \{0, 1\}$	rec: $x, y \in \{0, 1\}$	rec: $y{=}1 \Rightarrow x{=}1$	rec: $(y{=}1 \vee z{=}1) \Rightarrow x{=}1$	

Fig. 1. PARMv8 programs (top) and the possible values of x, y upon recovery (bottom); in all examples x and y are locations in persistent memory where $x \in X$, $y \notin X$, initially $x{=}y{=}0$, and thus $x, y \in \{0, 1\}$ after recovery.

observe $x{=}0 \wedge y{=}1$. This is indeed unsurprising as this behaviour is possible even during the normal (non-crashing) execution of the program in (P1).

However, the relaxed nature of the PARMv8 model allows for somewhat surprising behaviours that are not possible during normal executions. For instance, consider the program in Fig. 1b, corresponding to the program in the left thread of (P2) above. As discussed above, at no point during the execution of this program the $x{=}0 \wedge y{=}1$ behaviour is observable: the two writes cannot be reordered due to $\mathbf{DMB}_{\mathtt{full}}$. Nevertheless, in case of a crash it is possible under PARMv8 to observe $x{=}0 \wedge y{=}1$ after recovery. This is due to the relaxed persistency of PARMv8: the order in which writes are made visible to other threads (x before y) is separate from the order in which writes are persisted to memory (y before x). That is, the store and persist orders may disagree.[2]

In order to control the write-back of pending writes, the ARMv8 architecture provides an *explicit write-back* instruction, $\mathbf{wb}\ x$ [ARM 2018, p. C5-438].[3] The PARMv8 write-back instruction $\mathbf{wb}\ x$ persists all pending writes on all locations in the cache line of x. That is, when x is in the cache line X, written $x \in X$, the $\mathbf{wb}\ x$ instruction persists all pending writes on all locations $x' \in X$. A persist instruction $\mathbf{wb}\ x$ cannot be reordered with respect to earlier (in program order) writes on X; but may be reordered with respect to (both earlier and later) writes on non-X locations (locations not in X), As such, certain permitted reorderings mean that the write-back instructions may not take effect at the intended program point. Consider the example in Fig. 1c. Since the $y := 1$ write may be reordered before $x := 1; \mathbf{wb}\ x$, and the crash may occur after $y := 1$ (but before $x := 1; \mathbf{wb}\ x$), there is no guarantee upon recovery that $x := 1$ has persisted, *despite* the write-back instruction $\mathbf{wb}\ x$. It is therefore possible to observe $x{=}0 \wedge y{=}1$ upon recovery in Fig. 1c.

In order to afford more control over the order in which writes on different locations are persisted, the ARMv8 architecture provides *data synchronisation barriers*. A data synchronisation barrier, $\mathbf{DSB}_{\mathtt{full}}$, is strictly stronger than a $\mathbf{DMB}_{\mathtt{full}}$, and additionally awaits the completion of all previous write-back instructions. That is, (1) a write-back instruction cannot be reordered after a later (in program order) $\mathbf{DSB}_{\mathtt{full}}$ in the same thread; and (2) writes cannot be ordered before an earlier (in program order) $\mathbf{DSB}_{\mathtt{full}}$ in the same thread. For instance, consider the program in Fig. 1d obtained from that in Fig. 1c by introducing a $\mathbf{DSB}_{\mathtt{full}}$ after the write-back. Although in the case of Fig. 1c it is possible under PARMv8 to observe $y{=}1 \wedge x{=}0$ upon recovery as discussed above, the introduction of $\mathbf{DSB}_{\mathtt{full}}$ in the example above ensures that the write on x persists before that on y and thus upon recovery $y{=}1 \Rightarrow x{=}1$. More concretely, $\mathbf{wb}\ x$ cannot be reordered after $\mathbf{DSB}_{\mathtt{full}}$, the write on y cannot be reordered before $\mathbf{DSB}_{\mathtt{full}}$, and the execution of $\mathbf{DSB}_{\mathtt{full}}$ awaits the completion of $\mathbf{wb}\ x$. As such, if upon recovery $y{=}1$ (i.e. the write on y has executed and persisted prior to the crash), then $x{=}1$ (i.e. $\mathbf{wb}\ x$ and the write on x have also executed and persisted).

[2] Stores and persists are referred to as *point of coherency* (PoC) and *point of persistence* (PoP) in [ARM 2018, p. D4-2362].
[3] In [ARM 2018] this is referred to as **DC CVAP** or 'data or unified cache line clean by virtual address to point of persistence'; for brevity we write **wb** instead. The instruction was introduced in ARMv8.2 (released September 2017).

The examples discussed thus far all concern sequential programs and the persist orderings on the writes in the *same* thread. The example in Fig. 1e illustrates how persist orderings can be imposed on the writes of *different* threads. Note that the program in the left thread of Fig. 1e is that of Fig. 1d. As such, as before we have $y{=}1 \Rightarrow x{=}1$. Moreover, when the right thread in Fig. 1e reads 1 from y (written by the left thread), then under the ARMv8 model $y := 1$ is ordered before $\mathtt{a} := y$. As such, since $x := 1; \mathbf{wb}\ x; \mathbf{DSB}_{\mathtt{full}}$ is executed before $y := 1$ (as in Fig. 1d), $y := 1$ is ordered before $\mathtt{a} := y$, and $z := 1$ is ordered after $\mathtt{a} := y$ (due to the intervening $\mathbf{DMB}_{\mathtt{full}}$), we know $x := 1; \mathbf{wb}\ x; \mathbf{DSB}_{\mathtt{full}}$ is ordered before $z := 1$. Consequently, if upon recovery $z{=}1$ (i.e. $z := 1$ has persisted before the crash), then $x{=}1$ ($x := 1; \mathbf{wb}\ x$ must have also persisted before the crash). Note that by contrast $z{=}1 \Rightarrow y \in \{0, 1\}$. This is because $y := 1$ may persist after $z := 1$. As such, if a crash occurs after $z := 1$ has executed and persisted but before $y := 1$ has persisted, it is possible to observe $y{=}0, z{=}1$ after recovery, *even though* $y{=}0, z{=}1$ is never possible during normal (non-crashing) executions.

2.4 Language-Level Persistency: The PSER Model

With the emergence of non-volatile memory (NVM) technologies, researchers have identified *in-memory recoverable data structures* as one of the main applications of NVM. This is because, as we demonstrated in §2.3, NVM offers the durability of storage with the byte-addressability of RAM. This allows programmers to manipulate data structures directly using processor reads and writes, rather than high-latency software intermediaries such as the operating system or the file system. However, programmers rarely develop code on top of hardware models such as PARMv8. This is mainly because developing at this low-level (1) is significantly harder as it does not afford high-level abstractions such as encapsulation or concurrency control; and (2) requires an understanding of the hardware-specific instructions and guarantees, which in turn (3) hinders cross-platform portability. As such, researchers have proposed *language-level persistency* models instead [Gogte et al. 2018; Kolli et al. 2017], aiming to enable persistent programming in high-level languages such as C/C++. However, although the C++ committee has called for a study group to integrate NVM support into C++ in the near future [Douglas 2018], no mainstream programming language supports persistent programming as of yet.

One approach to language-level persistency is to extend a language such as C/C++ with write-back primitives analogous to **wb** x in PARMv8. This extension is simple in that it can be straightforwardly compiled into the corresponding write-backs in the underlying hardware. However, this approach is not conducive to simple programming as the overhead of ensuring correct persistency may render the code verbose. In particular, since the **wb** x instruction is *fine-grained* and specifies a single location (x) to be persisted, programmers need to continually keep a log of updated locations in order to issue the relevant write-backs and ensure correctness. This, however, is not an easy task for sophisticated data structures such as in-memory databases.

An alternative approach is to extend C/C++ with a more *coarse-grained* write-back instruction in the form of a *persistent barrier* [Izraelevitz et al. 2016b; Joshi et al. 2015; Pelley et al. 2014]. Rather than persisting the pending writes on a *single* location, a persistent barrier persists *all* pending writes. Persistent barriers are analogous to memory barriers: memory barriers order stores across multiple locations, while persistent barriers order persists across multiple locations. Persistent barriers thus allow programmers to control the write-back of several locations without keeping track of updated locations. However, implementing persistent barriers is not straightforward, and no existing architecture currently supports them. Indeed, following our conversations with engineers at ARM Research, we have been informed that such coarse-grained barriers are not part of their agenda in the foreseeable future as they are too costly to implement. We thus believe that architectures are unlikely to provide persistent primitives beyond those of fine-grained write-backs. As such, extending C/C++ with persistent barriers cannot be realised in a simple way as their

compilation to existing hardware (e.g. PARMv8) requires heavy code instrumentation to track the updated locations and to insert appropriate write-backs at compile time.

A third approach to providing language-level persistency is through *high-level persistent libraries*. Such libraries may be implemented using persistent primitives in existing hardware (e.g. **wb** x), while shielding their clients from such low-level details. Instead, they provide abstractions that make persistent programming much simpler. A powerful such abstraction is that of *atomic transactions*, used widely in the database community to ensure the consistency of persistent data. This has led the NVM community to study transactional implementations in the context of NVM [Avni et al. 2015; Kolli et al. 2016a; Shu et al. 2018; Tavakkol et al. 2018], most notably that of PMDK (persistent memory development kit) by Intel [2015], which provides a transactional persistence library.

NVM Transactions and TM/Database Transactions. Note that NVM transactions are more general than those of transactional memory (TM) in the shared memory concurrency literature, as well as those of database transactions in the distributed computing literature. In particular, TM transactions are typically run on *volatile* hardware. As such, TM semantics describe only the *consistency* guarantees during non-crashing executions and provide no persistency guarantees in case of a crash. By contrast, NVM transactions are run on persistent hardware and thus their semantics describe both consistency and persistency guarantees. On the other hand, although database transactions are run on traditionally persistent hardware (e.g. disks), there are two main differences between database and NVM transactions. First, the order in which database transactions are executed (made visible to other clients) is also the order in which they are persisted to hardware. That is, database transactions exhibit *strict* persistency, whilst NVM transactions may exhibit *strict or relaxed* persistency. Second, database transaction are persisted to hardware as soon as they are committed and made visible to other clients. That is, database transactions follow *unbuffered* (synchronous) persistency, whereas NVM transactions may follow *buffered or unbuffered* persistency.

To formalise the semantics of NVM transactions, we develop the *PSER* model. To our knowledge, PSER is the first formal transactional consistency and persistency model in the context of NVM.

The PSER Model. We develop the *PSER* (persistent serialisability) persistency model as an extension of the well-known transactional consistency model: *serialisability*. A *transaction* describes a block of code that executes *atomically*, ensuring that the transactional writes exhibit an all-or-nothing behaviour. Consider the transaction: **Tx** $[x := 1;\ \ y := 1]$. If initially $x{=}y{=}0$, at all points during the execution of **Tx** either $x{=}y{=}0$ or $x{=}y{=}1$. The most well-known transactional consistency model is *serialisability* (SER), where all concurrent transactions appear to execute atomically one after another in a total sequential order. Consider the transactional program (PTx) below:

$$\textbf{Tx1:}\begin{bmatrix} x := 1; \\ \mathsf{b} := y; \end{bmatrix} \;\Big\|\; \textbf{Tx2:}\begin{bmatrix} y := 1; \\ \mathsf{a} := x; \end{bmatrix} \tag{PTx}$$

Under serialisability, either **Tx1** executes before **Tx2** and thus $\mathsf{a}{=}1 \wedge \mathsf{b}{=}0$; or **Tx2** executes before **Tx1** and thus $\mathsf{a}{=}0 \wedge \mathsf{b}{=}1$. Serialisability is the gold standard of transactional consistency models, as it provides strong consistency guarantees with simple intuitive semantics.

We develop the PSER transactional persistency model by extending the atomicity and ordering guarantees of serialisability to persistency. That is, PSER provides (1) *persist atomicity*, ensuring that the persists in a transaction exhibit an all-or-nothing behaviour. For instance, if a crash occurs during the execution of **Tx** in the example above, upon recovery either $x{=}y{=}0$ or $x{=}y{=}1$. Moreover, PSER guarantees (2) *strict, buffered persistency* in that all concurrent transactions appear to persist atomically one after another in the same total sequential order in which they (appear to) have executed. As such, upon recovery, a *prefix* of the transactions in the total order may have persisted.

Basic domains

$\mathsf{a} \in \textsc{Reg}$ Registers
$v \in \textsc{Val}$ Values
$\tau \in \textsc{TId}$ Thread identifiers

Programs

$\mathsf{P} \in \textsc{Prog} \triangleq \textsc{TId} \xrightarrow{\text{fin}} \textsc{Com}$

Expressions and sequential commands

$\textsc{Exp} \ni \mathsf{e} ::= v \mid \mathsf{a} \mid \mathsf{e}{+}\mathsf{e} \mid \cdots$
$\textsc{PCom} \ni \mathsf{c} ::= \cdots$
$\textsc{Com} \ni \mathsf{C} ::= \mathsf{e} \mid \mathsf{c} \mid \textbf{let}\ \mathsf{a}{:=}\mathsf{C}\ \textbf{in}\ \mathsf{C}$
$\mid \textbf{if}\ (\mathsf{C})\ \textbf{then}\ \mathsf{C}\ \textbf{else}\ \mathsf{C} \mid \textbf{repeat}\ \mathsf{C}$

Fig. 2. A simple concurrent programming language

For example, consider the (PTx2) program below and assume that **Tx3** executes before **Tx4**.

$$\textbf{Tx3:}\begin{bmatrix} x := 1; \\ y := 1; \end{bmatrix} \Big\| \textbf{Tx4:}\begin{bmatrix} \mathsf{a} := x; \\ \textbf{if}\ \mathsf{a} > 0\ \textbf{then}\ z := 1; \end{bmatrix} \tag{PTx2}$$

If the execution of (PTx2) crashes, under PSER $z{=}1 \Rightarrow x{=}y{=}1$ upon recovery. That is, if **Tx4** has persisted ($z{=}1$), then the earlier transaction **Tx3** must have also persisted ($x{=}y{=}1$). As with serialisability, PSER provides strong consistency and persistency guarantees with intuitive semantics.

PSER Implementation. We present the formal semantics of PSER in §5. In order to show the feasibility of our PSER model, in §6 we develop a *sound* PSER implementation in PARMv8, thereby demonstrating that PSER correctly compiles to PARMv8.

3 A DECLARATIVE FRAMEWORK FOR PERSISTENCY SEMANTICS

We present a formal declarative framework for describing the persistency semantics of concurrent programs in the NVM context. In §3.1 we describe our programming language and its semantics; in §3.2 we present the necessary components for capturing the persistency guarantees of programs.

3.1 Programming Language and Semantics

Programming Language. To keep our presentation concise, we employ a simple concurrent programming language as given in Fig. 2. We assume a finite set $\textsc{Reg}$ of registers (local variables); a finite set $\textsc{Val}$ of values; a finite set $\textsc{TId} \subseteq \mathbb{N}^+$ of thread identifiers; and any standard interpreted language for expressions, $\textsc{Exp}$, containing at least registers and values. We use v as a metavariable for values, τ for thread identifiers, and e for expressions. The sequential fragment of the language is given by the $\textsc{Com}$ grammar and includes *primitive commands* (c), as well as the standard constructs of expressions, local variable assignment, conditionals and loops. The primitive commands in $\textsc{PCom}$ include model-specific instructions (e.g. reads and writes) and are thus determined by the underlying memory model. We model a multi-threaded program P as a function mapping each thread to its (sequential) program. We write $\mathsf{P} = \mathsf{C}_1 || \cdots || \mathsf{C}_n$ when $dom(\mathsf{P}) = \{\tau_1 \cdots \tau_n\}$ and $\mathsf{P}(\tau_i) = \mathsf{C}_i$. For better readability, we do not always follow syntactic conventions in our examples and write e.g. $\mathsf{a} := \mathsf{C}$ for **let** $\mathsf{a}{:=}\mathsf{C}$ **in** a, and $\mathsf{C}_1; \mathsf{C}_2$ for **let** $\mathsf{a}{:=}\mathsf{C}_1$ **in** C_2, where a is a fresh local variable.

Locations. Although non-volatile RAM is believed to eventually supplant volatile RAM completely, it may also be feasible to have hybrid memory hierarchies, where both volatile and non-volatile RAMs are combined together. To capture such hybrid hierarchies, we assume two distinct sets of *persistent memory locations*, $\textsc{PLoc}$, and *volatile memory locations*, $\textsc{VLoc}$ such that $\textsc{PLoc} \cap \textsc{VLoc} = \emptyset$. The set of *locations* is then given by $\textsc{Loc} \triangleq \textsc{PLoc} \cup \textsc{VLoc}$. We typically use $x_p, y_p, \ldots$ as meta-variables for persistent locations, and $x, y, \ldots$ for locations. We define the set of *cache lines* as $\textsc{CL} \triangleq \mathcal{P}(\textsc{Loc})$, and use $X, Y, \ldots$ as meta variables for cache lines. We write X_p to restrict the X cache line to its persistent locations: $X_p \triangleq X \cap \textsc{PLoc}$.

Labels and Events. In the literature of declarative models, the traces of shared memory accesses generated by a program are commonly represented as a set of execution graphs, where the graph nodes denote execution *events*, and graph edges capture the sundry relations on events. Each event corresponds to the execution of a primitive command ($c \in \textsc{PCom}$) and is a tuple of the form $e{=}\langle n, \tau, l\rangle$, where $n \in \mathbb{N}$ is the *event identifier* uniquely identifying e, $\tau \in \textsc{Tid}$ is the thread identifier of the executing thread, and $l \in \textsc{Lab}$ is the event *label*, described below.

As the set of primitive commands is memory model-specific, the set of event labels is consequently also model-specific. As such, we keep our framework parametric in the choice of memory model, and assume a set of *labels*, Lab, associated with the primitive commands of the underlying model. For instance, under the ARMv8 model, the command $\mathbf{DMB}_{\texttt{full}}$ is associated with the label $(\texttt{DMB}, \texttt{full})$. We further assume a set of *read labels*, RLab, and a set of *write labels*, WLab, such that $\textsc{RLab} \cup \textsc{WLab} \subseteq \textsc{Lab}$. The read and write labels are associated with (primitive) *read* and *write* commands, respectively. For instance, as we describe later in §4, under the ARMv8 model the (relaxed) write command $x := v$ is associated with the write label $(\texttt{W}, x, v, \texttt{rlx})$. We assume that functions $\texttt{loc}$, $\texttt{val}_\texttt{r}$ and $\texttt{val}_\texttt{w}$ respectively project the location, the read value and the written value of a label, where applicable. For instance, $\texttt{loc}(l){=}x$ and $\texttt{val}_\texttt{w}(l){=}v$ for $l{=}(\texttt{W}, x, v, \texttt{rlx})$. Finally, we assume a set of *durable labels*, $\textsc{DLab} \subseteq \textsc{Lab}$, associated with durable commands. Intuitively, durable commands are those whose effects may be observed when recovering from a crash. For instance, the effects of a write instruction $x_p := v$ may be observed upon recovery if the write of v on the persistent location x_p has persisted prior to the crash. As such, the label of $x_p := v$ is durable. Note that durability does not reflect whether the effects of the associated command *do persist*; rather that its effect *could persist*. That is, regardless of whether the effects of the write $x_p := v$ persist, its associated label is deemed durable. By contrast, a read instruction $a := x_p$ has no durable effects and its label is thus not durable. As write instructions on persistent locations are durable (their effects may be observed after a crash), we require that write labels on persistent locations be included in durable events: $\left\{l \in \textsc{WLab} \;\middle|\; \texttt{loc}(l) \in \textsc{PLoc}\right\} \subseteq \textsc{DLab}$. Moreover, we require that durable labels only include labels with persistent locations: $\textsc{DLab} \cap \left\{l \in \textsc{Lab} \;\middle|\; \texttt{loc}(l) \in \textsc{VLoc}\right\} = \emptyset$.

Parameter 1 (Labels)**.** Assume a set of *labels* Lab, a set of *read labels* $\textsc{RLab} \subseteq \textsc{Lab}$, and a set of *write labels* $\textsc{WLab} \subseteq \textsc{Lab}$. Assume functions $\texttt{loc} : \textsc{Lab} \rightharpoonup \textsc{Loc}$, $\texttt{val}_\texttt{r} : \textsc{RLab} \to \textsc{Val}$, and $\texttt{val}_\texttt{w} : \textsc{WLab} \to \textsc{Val}$. Assume a set of *durable labels*, $\textsc{DLab} \subseteq \textsc{Lab}$, such that $\left\{l \in \textsc{WLab} \;\middle|\; \texttt{loc}(l) \in \textsc{PLoc}\right\} \subseteq \textsc{DLab}$ and $\textsc{DLab} \cap \left\{l \in \textsc{Lab} \;\middle|\; \texttt{loc}(l) \in \textsc{VLoc}\right\} = \emptyset$.

Definition 1 (Events)**.** An *event* is a tuple $\langle n, \tau, l\rangle$, where $n \in \mathbb{N}$ is an event identifier, $\tau \in \textsc{Tid}$ is a thread identifier, and $l \in \textsc{Lab}$ is an event label.

We typically use a, b and e to range over events. The functions $\texttt{tid}$ and $\texttt{lab}$ respectively project the thread identifier and the label of an event. We lift the label functions $\texttt{loc}$, $\texttt{val}_\texttt{r}$ and $\texttt{val}_\texttt{w}$ to events, and given an event e, we write e.g. $\texttt{loc}(e)$ for $\texttt{loc}(\texttt{lab}(e))$.

Basic Executions. We define the semantics of programs in terms of *basic executions*. A basic execution, $G{=}\langle E, \mathrm{po}\rangle$, is a (partially) ordered set of events E (Def. 1), where po denotes the program order, describing whether one event precedes another in the control flow of the program. We write $G_0 \triangleq \langle \emptyset, \emptyset\rangle$ for the empty execution and $\{a\}_\mathrm{G} \triangleq \langle \{a\}, \emptyset\rangle$ for the execution with a single event a. Given two executions, $G_1{=}\langle E_1, \mathrm{po}_1\rangle$ and $G_2{=}\langle E_2, \mathrm{po}_2\rangle$, with disjoint sets of events ($E_1 \cap E_2{=}\emptyset$), we define their sequential composition, $G_1; G_2$, by ordering all G_1 events before those of G_2: $G_1; G_2 \triangleq \langle E_1 \cup E_2, \mathrm{po}_1 \cup \mathrm{po}_2 \cup (E_1 \times E_2)\rangle$. Similarly, we define their parallel composition, $G_1 \| G_2$, by placing no additional order between events of G_1 and G_2: $G_1 \| G_2 \triangleq \langle E_1 \cup E_2, \mathrm{po}_1 \cup \mathrm{po}_2\rangle$.

$$\mathsf{s} \in \textsc{Store} \triangleq \textsc{Reg} \to \textsc{Val} \qquad \textsc{RV} \triangleq \mathcal{P}\left((\{\bot\} \cup \textsc{Val}) \times \textsc{BExec}\right)$$
$$\langle\!|.|\!\rangle : \textsc{PCom} \to \textsc{Store} \to \textsc{RV} \qquad [\![.]\!] : \textsc{Com} \to \textsc{Store} \to \textsc{RV} \qquad \{\!|.|\!\} : \textsc{Prog} \to \textsc{RV}$$

$$[\![v]\!](\mathsf{s}) \triangleq \{\langle v, G_0\rangle\} \qquad [\![\mathsf{e}]\!](\mathsf{s}) \triangleq \{\langle \mathsf{s}(\mathsf{e}), G_0\rangle\} \qquad [\![\mathsf{C}]\!](\mathsf{s}) \triangleq \langle\!|\mathsf{C}|\!\rangle(\mathsf{s})$$

$$[\![\textbf{let } a{:=}\mathsf{C}_1 \textbf{ in } \mathsf{C}_2]\!](\mathsf{s}) \triangleq \left\{\langle r_2, G_1; G_2\rangle \;\middle|\; \langle v_1, G_1\rangle \in [\![\mathsf{C}_1]\!](\mathsf{s}) \wedge \langle r_2, G_2\rangle \in [\![\mathsf{C}_2]\!](\mathsf{s}[a \mapsto v_1])\right\} \cup \left\{\langle r_1, G_1\rangle \;\middle|\; \langle r_1, G_1\rangle \in [\![\mathsf{C}_1]\!](\mathsf{s}) \wedge \nexists v.\ r_1 = v\right\}$$

$$[\![\textbf{if } (\mathsf{C}) \textbf{ then } \mathsf{C}_1 \textbf{ else } \mathsf{C}_2]\!](\mathsf{s}) \triangleq \left\{\langle r_1, G; G_1\rangle \;\middle|\; \langle v, G\rangle \in [\![\mathsf{C}]\!](\mathsf{s}) \wedge v \neq 0 \wedge \langle r_1, G_1\rangle \in [\![\mathsf{C}_1]\!](\mathsf{s})\right\} \cup \left\{\langle r_2, G; G_2\rangle \;\middle|\; \langle v, G\rangle \in [\![\mathsf{C}]\!](\mathsf{s}) \wedge v = 0 \wedge \langle r_2, G_2\rangle \in [\![\mathsf{C}_2]\!](\mathsf{s})\right\} \cup \left\{\langle r, G\rangle \;\middle|\; \langle r, G\rangle \in [\![\mathsf{C}]\!](\mathsf{s}) \wedge \nexists v.\ r = v\right\}$$

$$[\![\textbf{repeat } \mathsf{C}]\!](\mathsf{s}) \triangleq \bigcup_{n\in\mathbb{N}} \left\{\langle 0, G_1; \ldots; G_n\rangle \;\middle|\; \begin{array}{l}\forall i < n.\ \langle v_i, G_i\rangle \in [\![\mathsf{C}]\!](\mathsf{s}) \wedge v_i{\neq}0 \\ \wedge \langle 0, G_n\rangle \in [\![\mathsf{C}]\!](\mathsf{s})\end{array}\right\} \cup \bigcup_{n\in\mathbb{N}} \left\{\langle \bot, G_1; \ldots; G_n\rangle \;\middle|\; \begin{array}{l}\forall i < n.\ \langle v_i, G_i\rangle \in [\![\mathsf{C}]\!](\mathsf{s}) \wedge v_i{\neq}0 \\ \wedge \langle -, G_n\rangle \in [\![\mathsf{C}]\!](\mathsf{s})\end{array}\right\}$$

$$\{\!|\mathsf{C}_1 \| \cdots \| \mathsf{C}_n|\!\} \triangleq \left\{\mathsf{par}(r_1, G_1, \cdots, r_n, G_n) \;\middle|\; \forall i \leq n.\ \langle r_i, G_i\rangle \in [\![\mathsf{C}_i]\!](\mathsf{s}_0)\right\}$$

$$\mathsf{par}(r_1, G_1, \cdots, r_n, G_n) \triangleq \begin{cases}\langle 1, G_1 \| \cdots \| G_n\rangle & \text{if } \exists v_1, \cdots, v_n \in \textsc{Val}.\ r_1{=}v_1 \wedge \cdots \wedge r_n{=}v_n \\ \langle \bot, G_1 \| \cdots \| G_n\rangle & \text{otherwise}\end{cases}$$

Fig. 3. The semantics of our programming language in Fig. 2

Definition 2 (Basic executions)**.** A *basic execution*, $G \in \textsc{BExec}$, is a tuple $G = \langle E, \mathrm{po}\rangle$, where E is a set of events with distinct identifiers (Def. 1) and $\mathrm{po} \subseteq E \times E$ is a strict partial order denoting the *program order* relation.

Semantics. Sequential commands are interpreted with respect to a *store* $\mathsf{s} \in \textsc{Store}$, which maps local variables (registers) to their values. The interpretation of a command C with respect to s, written $[\![\mathsf{C}]\!](\mathsf{s})$, generates a set of pairs of the form (r, G), where r denotes the *outcome* returned by C, and G denotes the corresponding basic execution leading to r. The outcome r may in turn be either $\bot$, when the computation has not yet terminated, or a value $v \in \textsc{Val}$.

The interpretation function $[\![.]\!]$ is given in Fig. 3, and is defined by induction over the language syntax. Interpreting value v yields outcome v with the empty execution G_0; interpreting an expression e evaluates e with respect to the store s, and thus returns outcome s(e) with empty execution G_0. Recall that primitive commands in PCom are model-specific and are supplied as a parameter to our framework. As such, we assume the existence of a primitive interpretation function, $\langle\!|.|\!\rangle$, that interprets a primitive command with respect to a store. The interpretation of a primitive command is then simply given by its primitive interpretation.

When $\langle r_1, G_1\rangle \in [\![\mathsf{C}_1]\!](\mathsf{s})$ and $\langle r_2, G_2\rangle \in [\![\mathsf{P}_2]\!](\mathsf{s})$, the interpretation of **let** $a{:=}\mathsf{C}_1$ **in** C_2 captures the sequential composition of C_1 and C_2 and comprises two cases depending on the outcome of C_1. When the computation of $[\![\mathsf{C}_1]\!](\mathsf{s})$ terminates (i.e. r_1 is a value), as expected the resulting outcome is that of P_2 (i.e. r_2) and the resulting execution is obtained from the sequential composition of executions $(G_1; G_2)$. On the other hand, when $[\![\mathsf{C}_1]\!](\mathsf{s})$ does not terminate, the interpretation yields $\langle r_1, G_1\rangle$. Analogously, the interpretation of a conditional is determined by the value of the condition.

Similarly, interpreting $[\![\textbf{repeat } \mathsf{C}]\!](\mathsf{s})$ comprises two cases. The first captures the case when the computation of $[\![\textbf{repeat } \mathsf{C}]\!](\mathsf{s})$ terminates after n iterations. That is, computing the first $n{-}1$ iterations of C yield non-zero values and thus do not trigger loop termination, whilst the n^{th}

iteration of C yields 0, indicating loop termination. As such, the loop is exited with return value 0. The resulting execution is that of the n iterations composed sequentially. The second captures the ongoing computation of $[\![\textbf{repeat}\ \mathsf{C}]\!](\mathsf{s})$ after n iterations, and thus the returned outcome is $\bot$. As before, the resulting execution is obtained from the sequential composition of the n executions accumulated thus far.

When $\langle r_i, G_i\rangle \in [\![\mathsf{C}_i]\!](\mathsf{s}_0)$ for $i \leq n$ and s_0 denotes the initial store which assigns 0 to all local variables, interpreting the program $\mathsf{C}_1 \| \cdots \| \mathsf{C}_n$ captures the parallel composition of $\mathsf{C}_1, \cdots, \mathsf{C}_n$ via the par function. The definition of par (at the bottom of Fig. 3) comprises two cases depending on the outcomes of constituent commands. When all computations terminate, the outcome is 1 (parallel composition does not return a meaningful value). Otherwise, the computation is marked as non-terminating ($\bot$). In both cases, the resulting execution is obtained from the parallel composition of the constituent executions $(G_1 \| \cdots \| G_n)$.

Parameter 2 (Primitive semantics)**.** Assume a primitive interpretation function, $\langle\!|.|\!\rangle : \textsc{Prog} \to \textsc{Store} \to \textsc{RV}$, that interprets a primitive command with respect to a store.

Definition 3 (Semantics)**.** The semantics of the programming language in Fig. 2 is given in Fig. 3.

3.2 Persistency Semantics

Notation. Given a relation r on a set A, we write $\mathsf{r}^?$, r^+ and r^* for the reflexive, transitive and reflexive-transitive closures of r, respectively. We write r^{-1} for the inverse of r; $\mathsf{r}|_A$ for $\mathsf{r} \cap (A \times A)$; $[A]$ for the identity relation on A, i.e. $\{(a,a) \mid a \in A\}$; $\mathsf{irreflexive}(r)$ for $\nexists a.\ (a,a) \in r$; and $\mathsf{acyclic}(\mathsf{r})$ for $\mathsf{irreflexive}(\mathsf{r}^+)$. We write $\mathsf{r}_1;\mathsf{r}_2$ for the relational composition of r_1 and r_2, i.e. $\{(a,b) \mid \exists c.\ (a,c) \in \mathsf{r}_1 \wedge (c,b) \in \mathsf{r}_2\}$. When r is a strict partial order, we write $\mathsf{r}|_{\mathsf{imm}}$ for the *immediate* edges in r, i.e. $\mathsf{r} \setminus (\mathsf{r};\mathsf{r})$. When A is a set of events, we define $A_x \triangleq \{a \in A \mid \mathtt{loc}(a) = x\}$, $A_X \triangleq \{a \in A \mid \mathtt{loc}(a) \in X\}$, and $A_p \triangleq \{a \in A \mid \mathtt{loc}(a) \in \mathrm{PLoc}\}$; i.e. A_p restricts A to events on persistent locations. Similarly, we define $\mathsf{r}_x \triangleq \mathsf{r} \cap (A_x \times A_x)$, $\mathsf{r}_X \triangleq \mathsf{r} \cap (A_X \times A_X)$, and $\mathsf{r}_p \triangleq \mathsf{r} \cap (A_p \times A_p)$.

Executions and Chains. The traces of shared memory accesses generated by a program are commonly represented as a set of *executions*, where each execution G is a graph comprising: (i) a set of events denoting the graph nodes; and (ii) a number of relations on events, denoting the sundry graph edges. It is common practice to consider *complete* executions only, i.e. those that do not crash. However, this assumption renders this model unsuitable for capturing the crashing behaviour of executions in the presence of persistent memory. Instead, we model an *execution chain* C as a sequence $G_1, \cdots, G_n$, with each G_i describing an execution *era* between two adjacent crashes. More concretely, when an execution of program P crashes $n{-}1$ times, we model this as the chain $C{=}G_1, \cdots, G_n$, where (1) G_1 describes the initial era between the start of execution up to the first crash; (2) for all $i \in \{2, \cdots, n{-}1\}$, G_i denotes the i^{th} execution era, recovering from the $(i{-}1)^{\text{st}}$ crash; and (3) G_n describes the final execution era terminating successfully.

Definition 4 (Executions)**.** An *execution*, $G \in \textsc{Exec}$, is a tuple $(E, I, P, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{nvo})$, where:

- E denotes a set of *events*. The set of *read* events in E is denoted by $R \triangleq \left\{e \in E \,\middle|\, \mathtt{lab}(e) \in \textsc{RLab}\right\}$; the set of *write* events, W, and *durable* events, D, are defined analogously.
- I is a set of *initialisation events*, comprising a single write event $w_x \in W$ for each location x.
- P is a set of *persisted events* such that $I_p \subseteq P \subseteq D$.
- $\mathsf{po} \subseteq E \times E$ denotes the *'program-order' relation*, defined as a disjoint union of strict total orders, each ordering the events of one thread, with $I \times (E \setminus I) \subseteq \mathsf{po}$.
- $\mathsf{rf} \subseteq W \times R$ denotes the *'reads-from' relation* between write and read events of the same location with matching values; i.e. $(a,b) \in \mathsf{rf} \Rightarrow \mathtt{loc}(a){=}\mathtt{loc}(b) \wedge \mathtt{val}_{\mathtt{w}}(a){=}\mathtt{val}_{\mathtt{r}}(b)$. Moreover, rf is total and functional on its range, i.e. every read is related to exactly one write.

- $\mathsf{mo} \subseteq E \times E$ is the *'modification-order'*, denoting a strict partial order defined as the disjoint union of relations $\{\mathsf{mo}_x\}_{x \in \mathrm{Loc}}$, such that each mo_x is a strict total order on W_x and $I_x \times (W_x \setminus I_x) \subseteq \mathsf{mo}_x$.
- $\mathsf{nvo} \subseteq D \times D$ is the *'non-volatile-order'*, defined as a strict total order on D, such that $I_p \times (D \setminus I_p) \subseteq \mathsf{nvo}$ and $dom(\mathsf{nvo}; [P]) \subseteq P$.

In the context of an execution graph G – we often use the "G." prefix to make this explicit – the persisted events P include those durable events ($P \subseteq D$) whose effects have reached the persistent memory; e.g. those stores that have persisted. As such, persisted events include initialisation writes on persistent locations ($I_p \subseteq P$). The 'modification-order' mo constrains the visible order of stores to the memory (i.e. allowable visible memory states) between threads. Analogously, the 'non-volatile-order' nvo constrains the visible order in which stores are committed to the persistent memory. Note that nvo prescribes the order in which writes are persisted to memory. As such, we require that the persisted events in P be downward-closed with respect to nvo: $dom(\mathsf{nvo}; [P]) \subseteq P$. That is, let $e_1 \cdots e_n$ denote an enumeration of D according to nvo (since nvo describes a total order on D); as P is downward-closed with respect to nvo, we know there exists $1 \leq i \leq n$ such that $e_1, \cdots, e_i \in P$ and $e_{i+1}, \cdots, e_n \in D \setminus P$.

For simplicity, we assume that event identifiers in each thread are ordered by po: for all a, b: $(a, b) \in \mathsf{po}|_{\mathrm{imm}} \iff \mathtt{id}(b) = \mathtt{id}(a) + 1$. Lastly, we define the '*reads-before*' relation as: $\mathsf{rb} \triangleq (\mathsf{rf}^{-1}; \mathsf{mo}) \setminus \mathrm{id}$, relating a read r to all writes w that are mo-after the write r reads from.

Note that in this initial stage, executions are unrestricted in that there are few constraints on rf, mo and nvo. Such restrictions are determined by the set of memory-model specific *consistent* executions. We thus assume a *consistency predicate*, $\mathrm{cons}_{\mathcal{M}}(.)$, which determines whether an execution is $\mathcal{M}$-consistent (i.e. consistent under the $\mathcal{M}$ memory model). In the upcoming sections we define execution consistency for the PARMv8 and PSER models discussed in §2.

Parameter 3 (Consistency)**.** Assume a *consistency predicate*, $\mathrm{cons}_{\mathcal{M}}(.) : \textsc{Exec} \to \{\mathsf{true}, \mathsf{false}\}$.

Definition 5 (Chains)**.** A *chain* C is a sequence $G_1, \cdots, G_n$ of executions such that for $1 \leq i < n$ and $G_i = (E_i, I_i, P_i, \mathsf{po}_i, \mathsf{rf}_i, \mathsf{mo}_i, \mathsf{nvo}_i)$:

- $\forall x \in \mathrm{Loc}.\ \exists w.\ w \in I_1 \wedge \mathtt{loc}(w) = x \wedge \mathtt{val_w}(w) = 0$;
- $\forall x \in \mathrm{VLoc}.\ \exists w.\ w \in I_{i+1} \wedge \mathtt{loc}(w) = x \wedge \mathtt{val_w}(w) = 0$;
- $\forall x \in \mathrm{PLoc}.\ \exists w, e.\ w \in I_{i+1} \wedge \mathtt{loc}(w) = x \wedge e = \max\left(\mathsf{nvo}_i|_{P_i \cap W_x}\right) \wedge \mathtt{val_w}(w) = \mathtt{val_w}(e)$;
- $P_n = E_n \cap D$.

Given a memory model $\mathcal{M}$, a chain $C = G_1, \cdots, G_n$ is *$\mathcal{M}$-valid* if $\mathrm{cons}_{\mathcal{M}}(G_i)$ holds for all G_i.

The first axiom ensures that in the first era all locations are initialised with 0. The second axiom ensures that in each subsequent $(i{+}1)^{\mathrm{st}}$ era all *volatile* locations are initialised with 0; That is, volatile locations lose their values upon a crash and are thus reset to 0 in the following era. By contrast, the third axiom ensures that in each subsequent $(i{+}1)^{\mathrm{st}}$ era all *persistent* locations are initialised with a value persisted by a write in the previous (i^{th}) era maximally (in nvo_i). In other words, persistent locations retain their values in case of a crash. The last axiom ensures that the final era executes completely (does not crash) by stipulating that all its durable events be persisted. That is, in the absence of a crash, all durable events are eventually persisted.

Persistent Programs. In the persistent setting, each program P is associated with a *recovery mechanism* describing the code to be executed upon recovery from a crash. As such, we model a *persistent program*, $\mathbb{P} \in \textsc{PProg}$, as a pair $\langle \mathsf{P}, \mathtt{rec} \rangle$, where $\mathsf{P} \in \textsc{Prog}$ denotes the original program to be executed, and rec denotes its recovery mechanism. A naive recovery mechanism of P may always choose to *restart* the execution of P from the beginning. However, a more sophisticated mechanism may *resume* the execution of P upon recovery by determining the progress made prior

to the crash. To do this, the recovery mechanism may inspect the memory to identify the operations whose effects have persisted to memory, and then resume P from the last persisted operation in each thread. To capture this, we model a recovery mechanism as a function $\texttt{rec} : \textsc{Prog} \times \textsc{Exec} \to \textsc{Prog}$. That is, $\texttt{rec}(\mathsf{P}, G)$ describes the recovery mechanism associated with P when the memory upon recovery is that obtained after execution G. Intuitively, execution G corresponds to the previous execution era and thus the state of memory upon recovery can be ascertained by inspecting G.

Definition 6 (Persistent programs)**.** A *recovery mechanism* is a function $\texttt{rec} \in \textsc{Rec} : \textsc{Prog} \times \textsc{Exec} \to \textsc{Prog}$. A *persistent program* is a pair $\mathbb{P} \in \textsc{PProg} \triangleq \textsc{Prog} \times \textsc{Rec}$.

From Programs to Executions and Chains. Recall from §3.1 that we described the semantics of a program P as a set of *basic executions* (Def. 2) associated with P. The set of executions associated with P, written $\texttt{exec}(\mathsf{P})$, contains those executions (Def. 4) that can be projected to basic executions that have terminated. That is, $G \in \texttt{exec}(\mathsf{P})$ iff there exists v such that $\langle v, \langle G.E, G.\mathrm{po}\rangle\rangle \in \{\!|\mathsf{P}|\!\}$. The set of partial executions associated with P, written $\texttt{pexec}(\mathsf{P})$, contains execution prefixes. Intuitively, partial executions contain those executions that are rendered incomplete due to a crash.

Analogously, in the persistent setting each persistent program $\mathbb{P}$ is a associated with a set of *chains*. More concretely, a persistent program $\mathbb{P}{=}\langle \mathsf{P}, \texttt{rec}\rangle$ is associated with a chain $C{=}G_1, \cdots, G_n$ if: (1) G_1 is a partial execution of P; (2) G_i is a partial execution of $\texttt{rec}(\mathsf{P}, G_{i-1})$, for all $2 \leq i \leq n{-}1$; and (3) G_n is an execution of $\texttt{rec}(\mathsf{P}, G_{n-1})$. Note that executions of all but the last era are partial in that they have failed to run to completion due to a crash.

Definition 7 (Program executions)**.** Given a program P, the set of *executions of* P, written $\texttt{exec}(\mathsf{P})$, is: $\left\{G \in \textsc{Exec} \;\middle|\; \exists v \in \textsc{Val}.\ \langle v, \langle G.E, G.\mathrm{po}\rangle\rangle \in \{\!|\mathsf{P}|\!\}\right\}$; the set of *partial executions of* P, written $\texttt{pexec}(\mathsf{P})$, is: $\left\{G \in \textsc{Exec} \;\middle|\; \exists \langle -, \langle E', \mathrm{po}'\rangle\rangle \in \{\!|\mathsf{P}|\!\}.\ \langle G.E, G.\mathrm{po}\rangle \sqsubseteq \langle E', \mathrm{po}'\rangle\right\}$, where:

$$\langle E, \mathrm{po}\rangle \sqsubseteq \langle E', \mathrm{po}'\rangle \stackrel{\text{def}}{\iff} E \subseteq E' \wedge \mathrm{po} \subseteq \mathrm{po}' \wedge \mathit{dom}(\mathrm{po}'; [E]) \subseteq E$$

Definition 8 (Program chains)**.** Given a persistent program $\mathbb{P}{=}\langle \mathsf{P}, \texttt{rec}\rangle$, the *set of chains* associated with $\mathbb{P}$, written $\texttt{chain}(\mathbb{P})$, contains chains of the form $G_1, \cdots, G_n$ such that: (1) $G_1 \in \texttt{pexec}(\mathsf{P})$; (2) $G_i \in \texttt{pexec}(\texttt{rec}(\mathsf{P}, G_{i-1}))$ for $2 \leq i \leq n{-}1$; and (3) $G_n \in \texttt{exec}(\texttt{rec}(\mathsf{P}, G_{n-1}))$.

4 THE PERSISTENT ARMv8 MODEL (PARMv8)

We present the formal PARMv8 model declaratively as an instance of the general framework in §3.

PARMv8 Programming Language. The *PARMv8 programming language* is the programming language in Fig. 2 instantiated with *PARMv8 primitive commands* given by the grammar below:

$$\begin{array}{lll} \textsc{PCom}_{\textsc{PARMv8}} \ni \mathsf{c} ::= \mathbf{load}_{lm}(x) \mid \mathbf{store}_{sm}(x, \mathsf{e}) \mid \mathbf{CAS}_{lm,sm}(x, \mathsf{e}, \mathsf{e}') \mid \mathbf{DMB}_{bm} \mid \mathbf{DSB}_{bm} \mid \mathbf{wb}\ x_p & & \\ \text{where} \quad lm ::= \texttt{rlx} \mid \texttt{A} \mid \texttt{Q} & sm ::= \texttt{rlx} \mid \texttt{L} \mid \texttt{Q} & bm ::= \texttt{ld} \mid \texttt{st} \mid \texttt{full} \end{array}$$

The highlighted sections denote the persistent extensions from the original ARMv8 model by Pulte et al. [2018]. The $\mathbf{load}_{lm}(x)$ instruction denotes a load (read) from x with the *load mode lm*. Analogously, the $\mathbf{store}_{sm}(x, \mathsf{e})$ instruction denotes a store (write) on x with the *store mode sm*. The $\mathbf{CAS}(x, \mathsf{e}, \mathsf{e}')$ denotes an atomic 'compare-and-swap' (with the given load and store modes), where the value of location x is compared against e: if the values match then the value of x is set to e' and 1 is returned; otherwise x is left unchanged and 0 is returned. The $\mathbf{DMB}_{bm}$ denotes a 'data memory barrier' with the barrier mode *bm*. Load, store and barrier modes vary in strength and accordingly alter the imposed ordering constraints. Understanding these constraints is not necessary for understanding the persistency semantics of PARMv8. We refer the reader to [Pulte et al. 2018] for a description of these guarantees. As before, for readability we write $a :=_{lm} x$ for

let a:=**load**$_{lm}(x)$ **in** a when a is fresh, and write $x :=_{sm} e$ for **store**$_{sm}(x, e)$. We omit the access modes when relaxed (rlx) and simply write $x := e$ and $a := x$ for $x :=_{\texttt{rlx}} e$ and $a :=_{\texttt{rlx}} x$, respectively.

To capture the PARMv8 persistency semantics, we extend the ARMv8 language with **DSB**$_{bm}$ and **wb** x_p, denoting 'data synchronisation barrier' and 'write-back' instructions, respectively. As discussed in §2, a **DSB**$_{bm}$ is strictly stronger than a **DMB**$_{bm}$ (with the same access mode), and may additionally constrain the order in which writes are *persisted*, as we discuss shortly. Recall that **wb** x commits all pending writes on x to persistent memory. As such, write-back instructions are only issued on persistent locations since writes on volatile locations cannot be persisted.

PARMv8 Labels. As discussed in §3, event labels are associated with the primitive commands of the underlying memory model. Given access modes *lm*, *sm* and *bm* above, a PARMv8 label is either: a *load* label $(\texttt{R}, x, v, lm)$, a *store* label $(\texttt{W}, x, v, sm)$, an *update* (CAS) label $(\texttt{U}, x, v, v', lm, sm)$, a *DMB* label $(\texttt{DMB}, bm)$, a *DSB* label $(\texttt{DSB}, bm)$, or a *write-back* label $(\texttt{WB}, x_p)$. *PARMv8 read* labels comprise load and update labels; *PARMv8 write labels* comprise store and update labels. *PARMv8 durable labels* comprise write-back labels, as well as write labels on persistent locations. The $\texttt{loc}, \texttt{val}_\texttt{r}, \texttt{val}_\texttt{w}$ functions are defined as expected.

Definition 9 (PARMv8 labels)**.** The set of *PARMv8 labels* is defined as follows:

$$\textsc{Lab}_{\textsc{PARMv8}} \triangleq \left\{ \begin{array}{l} (\texttt{R}, x, v, lm), (\texttt{W}, x, v, sm), (\texttt{U}, x, v, v', lm, sm), \\ (\texttt{DMB}, bm), (\texttt{DSB}, bm), (\texttt{WB}, x_p) \end{array} \middle| \begin{array}{l} x \in \textsc{Loc} \wedge v, v' \in \textsc{Val} \\ x_p \in \textsc{PLoc} \end{array} \right\}$$

PARMv8 read labels are: $\textsc{RLab}_{\textsc{PARMv8}} \triangleq \textsc{Lab}_{\textsc{PARMv8}} \cap \{(\texttt{R}, x, v, lm), (\texttt{U}, x, v, v', lm, sm)\}$; *PARMv8 write labels* are: $\textsc{WLab}_{\textsc{PARMv8}} \triangleq \textsc{Lab}_{\textsc{PARMv8}} \cap \{(\texttt{W}, x, v, sm), (\texttt{U}, x, v, v', lm, sm)\}$; *PARMv8 durable labels* are: $\textsc{DLab}_{\textsc{PARMv8}} \triangleq \textsc{Lab}_{\textsc{PARMv8}} \cap \{(\texttt{WB}, x), (\texttt{W}, x_p, v, sm) \mid x_p \in \textsc{PLoc}\}$.

PARMv8 Executions. A *PARMv8 event* is an event (Def. 1) with a PARMv8 label; a *PARMv8 execution* is an execution (Def. 4) with PARMv8 events. The sets of read (R), write (W) and durable (D) events are as in Def. 4. The sets of write-back (WB), DMB (DMB_{bm}) and DSB (DSB_{bm}) events are defined analogously. For instance, we write $DSB_{\texttt{full}}$ for DSB events with the full barrier mode.

Compared to our executions (Def. 4), the ARMv8 executions of [Pulte et al. 2018] carry additional components that record address, data and control dependencies between events, which in turn impose additional ordering constraints. Understanding the details of these dependencies is not necessary for understanding the PARMv8 persistency semantics. We refer the reader to [Podkopaev et al. 2019] for the details of how these dependencies are computed for a given program.

Definition 10 (PARMv8 consistency)**.** A PARMv8 execution $(E, I, P, \mathrm{po}, \mathrm{rf}, \mathrm{mo}, \mathrm{nvo})$ is *PARMv8-consistent* iff:

- ARMv8 axioms in [Pulte et al. 2018] hold with DMB_{bm} replaced with $DMB_{bm} \cup DSB_{bm}$ (ARM)
- $(\mathrm{po}^?; [DMB_{\texttt{full}} \cup DSB_{\texttt{full}}]; \mathrm{po}^?) \setminus \mathrm{id} \subseteq \mathrm{ob}$ (ARM-OB-BAR)
- $\forall X \in \mathrm{CL}.\ [W_X \cup R_X]; \mathrm{po}; [WB_X] \subseteq \mathrm{ob}$ (OB-W-WB)
- $\forall X \in \mathrm{CL}.\ [WB_X]; \mathrm{po}; [WB_X] \subseteq \mathrm{ob}$ (OB-WB-WB)
- $dom([WB]; \mathrm{ob}; [DSB_{\texttt{full}}]) \subseteq P$ (NVO-PERS)
- $[WB]; \mathrm{ob}; [DSB_{\texttt{full}}]; \mathrm{ob}; [D] \subseteq \mathrm{nvo}$ (NVO-WB-D)
- $\forall X \in \mathrm{CL}.\ [W_{X_p}]; \mathrm{ob}; [WB_{X_p}] \subseteq \mathrm{nvo}$ (NVO-W-WB)
- $\forall x_p \in \textsc{PLoc}.\ \mathrm{mo}_{x_p} \subseteq \mathrm{nvo}$ (NVO-MO)

The ARMv8 model by Pulte et al. [2018] defines the permitted program behaviours by constraining the visible order of memory instructions. To do this, they define the 'ordered-before' relation, ob, which prescribes the ordering constraints that must be preserved. That is, ob denotes the 'volatile-memory-order' and thus two ob-related instructions cannot be reordered. However, note that

the axioms of Pulte et al. [2018] do not account for **DSB** and **wb** instructions; nor have they any bearing on the PARMv8 persistency semantics as they impose no constraints on nvo. We capture the ordering constraints of **DSB** and **wb** via (ARM), (OB-W-WB) and (OB-WB-WB). We describe the PARMv8 persistency semantics via (NVO-PERS), (NVO-W-WB), (NVO-WB-D) and (NVO-MO).

The (ARM) axiom imports the ARMv8 axioms by Pulte et al. [2018] with DMB_{bm} replaced with $DMB_{bm} \cup DSB_{bm}$. This is because **DSB** barriers are strictly stronger than **DMB** ones (with the same mode). This is captured by the following text in the ARM reference manual, where *italicised text* in square brackets denotes our added clarification:

> "it *[***DSB***]* acts as a stronger barrier than a **DMB** and ordering that is created by a **DMB** with specific options is also generated by a **DSB** with the same options." [ARM 2018, p. B2-106]

For brevity, we have elided the axioms included in (ARM), with the exception of the (ARM-OB-BAR) axiom which we repeat here. This is because apart from (ARM-OB-BAR), the axioms in (ARM) have no bearing on the persistency behaviour of PARMv8 programs; we refer the reader to [Pulte et al. 2018] for the remaining axioms in (ARM). The (ARM-OB-BAR) axiom states that if two instructions are po-ordered and are separated by a full barrier (a $\mathbf{DMB}_{\mathtt{full}}$ or $\mathbf{DSB}_{\mathtt{full}}$) instruction, then they are also ob-related and thus cannot be reordered.

The (OB-W-WB) and (OB-WB-WB) axioms describe the ordering constraints on **wb** instructions: given locations x, x' in the same cache line X, a write-back event wb on x, and a write/read/write-back event e on x', if e is po-before wb, then e is executed before (ob-before) wb. This reflects the following text in the ARM manual (see Remark 1 below for a clarification of 'normal' memory):

> "All data cache instructions … that specify an address *[including* **wb***]*:
> - Execute in program order relative to loads *[reads]* or stores *[writes]* which access an address in normal memory … within the same cache line …
> - Execute in program order relative to other data cache instructions *[including* **wb***]* … that specify an address within the same cache line …
> - Can execute in any order relative to loads *[reads]* or stores *[writes]* that access an address in a different cache line … unless a $\mathbf{DMB}_{\mathtt{full}}$ or $\mathbf{DSB}_{\mathtt{full}}$ is executed between the instructions.
> - Can execute in any order relative to other data cache instructions *[including* **wb***]* … that specify an address in a different cache line … unless a $\mathbf{DMB}_{\mathtt{full}}$ or $\mathbf{DSB}_{\mathtt{full}}$ is executed between the instructions." [ARM 2018, p. D4-2371]

The (NVO-PERS) axiom states that a write-back instruction executed before (ob-before) a $\mathbf{DSB}_{\mathtt{full}}$ is always *persisted* and is thus included in P. This is because executing a $\mathbf{DSB}_{\mathtt{full}}$ instruction awaits the completion of all previously executed **wb** instructions:

> "a $\mathbf{DSB}_{\mathtt{full}}$ instruction will not complete until all previous **wb** instructions have completed" [ARM 2018, p. D4-2366]

The (NVO-WB-D) axiom describes the persist orderings imposed by $\mathbf{DSB}_{\mathtt{full}}$. More concretely, recall that a $\mathbf{DSB}_{\mathtt{full}}$ instruction awaits the completion (persistence) of all its ob-before write-backs (NVO-PERS). As such, all instructions executed after (ob-after) a $\mathbf{DSB}_{\mathtt{full}}$ are persisted after (nvo-after) the write-backs ob-before the $\mathbf{DSB}_{\mathtt{full}}$.

Recall from §2.3 that when $x \in X$, then executing **wb** x persists all pending writes on X to persistent memory. That is, the effect of **wb** x is committed to persistent memory (i.e. **wb** x persists) once all pending (ob-earlier) writes on X have persisted. As such, all pending writes on X persist before **wb** x and are thus nvo-ordered before it. This is captured by the (NVO-W-WB) axiom. Note that (ARM-OB-BAR), (OB-W-WB), (NVO-WB-D) and (NVO-W-WB) together ensure the recovery behaviour illustrated in the examples of Fig. 1d and Fig. 1e in §2.3.

Lastly, the (NVO-MO) axiom states that for each location x, the volatile and non-volatile memory orders (mo and nvo) agree. This is illustrated in the following examples:

$x := 1$;			**if** $z = 1$ **then**	
$x := 2$;		$x := 1$;	$x := 2$;	
wb x;	(P3)	**DMB**$_{\texttt{full}}$;	**wb** x;	(P4)
DSB$_{\texttt{full}}$;		$z := 1$	**DSB**$_{\texttt{full}}$;	
$y := 1$			$y := 1$	
rec: $y{=}1 \Rightarrow x{=}2$		rec: $y{=}1 \Rightarrow x{=}2$		

As po and mo orders agree under ARMv8, in (P3) $x := 1$ is mo-ordered before $x := 2$. Therefore, from (NVO-MO) we know $x := 1$ is also nvo-ordered before $x := 2$. Note that the program in (P3) is that in Fig. 1d with the additional $x := 2$ write on x. As such, as in Fig. 1d, if upon recovery $y := 1$ has persisted, then the nvo-latest write on x, i.e. $x := 2$, must also have persisted, and thus $y{=}1 \Rightarrow x{=}2$.

The program in (P4) illustrates a similar scenario with mo between the writes in *different* threads. When the **if** condition holds (i.e. the second thread reads from the first thread), then under ARMv8 $x := 1$ is mo-ordered before $x := 2$. Consequently, as in (P3) we have $y{=}1 \Rightarrow x{=}2$ after recovery.

Remark 1 (Non-cacheable locations)**.** The ARM manual [ARM 2018] distinguishes between *normal* (cacheable) and *non-cacheable* locations, with the latter denoting those locations whose contents cannot be cached locally and must be accessed directly from memory. The existing ARMv8 specification by Pulte et al. [2018] does not model non-cacheable locations and only describes the behaviour of normal locations. As we model PARMv8 as an extension of ARMv8, we also model normal locations only. That is, we assume all locations in PARMv8 are normal (cacheable).

The interplay between the ARMv8 *consistency* axioms of Pulte et al. [2018] in (ARM) above and non-cacheable locations is rather subtle and is beyond the scope of this paper. However, it is straightforward to alter our extended axioms above to model non-cacheable locations. In particular, while (OB-WB-WB), (NVO-PERS), (NVO-WB-D), (NVO-W-WB) and (NVO-MO) axioms remain unchanged, the (OB-W-WB) axiom must be altered to refer to normal locations only. This is because the ordering constraints of write-backs on non-cacheable locations are weaker than those on normal locations. Specifically, a write-back on a non-cacheable location x is ob-ordered with respect to a po-earlier read or write on x *only if* the two are separated by a **DMB** or a **DSB**:

> "All data cache instructions … that specify an address *[including* **wb***]*:
> - Can execute in any order relative to loads *[reads]* or stores *[writes]* that access any address … with Inner Non-cacheable attribute unless a **DMB**$_{\texttt{full}}$ or **DSB**$_{\texttt{full}}$ is executed between the instructions." [ARM 2018, p. D4-2371]

In other words, if location x in Fig. 1d of §2.3 were non-cacheable, in order to ensure that upon recovery $y{=}1 \Rightarrow x{=}1$, one would need to insert an additional **DMB**$_{\texttt{full}}$ (or **DSB**$_{\texttt{full}}$) instruction between the write on x and its write-back, as described in the ARM manual [ARM 2018, p. D4-2366]:

$$x := 1; \quad \textbf{DMB}; \quad \textbf{wb}\ x; \quad \textbf{DSB}_{\texttt{full}}; \quad y := 1$$

Remark 2 (PARMv8 Fidelity)**.** We have discussed our PARMv8 model and examples at length with the engineers at ARM Research, and we have been reassured that PARMv8 faithfully describes the persistency semantics of the subset of ARM instructions modelled.

5 THE PERSISTENT SERIALISABILITY MODEL (PSER)

We present the formal declarative PSER model as an instance of the general framework in §3. To show the utility of PSER, we demonstrate that PSER can be used to convert *any* correct *sequential* implementation of a given library $\mathcal{L}$ to a *correct, concurrent and persistent* implementation of $\mathcal{L}$

(§5.1). To show the feasibility of PSER, later in §6 we develop a sound PSER implementation in PARMv8, thus demonstrating that PSER correctly compiles to PARMv8.

PSER Programming Language. The *PSER primitive commands* comprise simple load (read) and store (write) operations, and are given by the grammar below:

$$\textsc{Com}_{\textsc{pser}} \ni \mathsf{c} ::= \mathbf{load}(x) \mid \mathbf{store}(x_p, \mathsf{e})$$

Note that we only allow stores on persistent locations. As we describe shortly, this allows us to keep the PSER model simple with respect to *transactional persist atomicity*. As before, we write $a := x$ for **let** a:=**load**(x) **in** a (when a is fresh), and write $x_p := \mathsf{e}$ for **store**(x_p, e). For simplicity, we assume that the (sequential) programs in each thread comprise a sequence of PSER *transactions*. That is, we define the set of *PSER programs*, $\textsc{Prog}_{\textsc{pser}} \subseteq \textsc{Prog}$, as follows where C denotes a sequential program as defined in Fig. 2, instantiated with PSER primitive commands:

$$\textsc{Prog}_{\textsc{pser}} \ni \mathsf{P} ::= \textsc{Tid} \xrightarrow{\text{fin}} \textsc{Com}_{\textsc{pser}} \qquad \textsc{Com}_{\textsc{pser}} \ni \mathsf{C}_{\textsc{pser}} ::= [\mathsf{C}] \mid \mathsf{C}_{\textsc{pser}}; \mathsf{C}_{\textsc{pser}}$$

PSER Labels and Events. In order to distinguish the events of one transaction from another, we assume a finite set of *transaction identifiers*, TXID, ranged over by ξ. We model a transaction identifier as a pair $\xi = \langle \tau, n \rangle$, where τ denotes the thread with which the transaction is associated, and n identifies the transaction within thread τ. A PSER label is then either: (1) a *load* label $(\mathtt{R}, x, v, \xi)$, for a primitive load from x in ξ; or (2) a *store* label $(\mathtt{W}, x_p, v, \xi)$, for a primitive store to x_p in ξ; or (3) a *begin* label $(\mathtt{B}, \xi)$, marking the beginning of ξ; or (4) an *end* label $(\mathtt{E}, \xi)$, marking the end of ξ. *PSER read* and *write labels* comprise load and store labels, respectively. *PSER durable labels* coincide with PSER write labels. Functions $\mathtt{loc}, \mathtt{val}_\mathtt{r}, \mathtt{val}_\mathtt{w}$ are defined as expected. The function $\mathtt{tx}$ returns the transaction identifier of a PSER label. A *PSER event* is an event (Def. 1) with a PSER label. As before, we lift the $\mathtt{tx}$ function to events and write e.g. $\mathtt{tx}(e)$ for $\mathtt{tx}(\mathtt{lab}(e))$.

Given an execution G, the *'same-transaction' relation*, $\mathsf{st} \in G.E \times G.E$, is the equivalence relation given by $\mathsf{st} \triangleq \{(a,b) \in G.E \times G.E \mid \mathtt{tx}(a){=}\mathtt{tx}(b)\}$. Given a relation r on $G.E$, we write r_T for *lifting* r to (equivalence) classes: $\mathsf{r}_\mathsf{T} \triangleq \mathsf{st}; (\mathsf{r} \setminus \mathsf{st}); \mathsf{st}$. We write $[a]_\mathsf{st}$ for the st class that contains a, i.e. $[a]_\mathsf{st} \triangleq \{e \in G.E \mid (a,e) \in \mathsf{st}\}$. Note that a class without an end event denotes a transaction whose execution was rendered *incomplete* by a crash. We write $G.T$ for the events of *complete transactions* in G; i.e. those events whose associated end events is in G: $G.T \triangleq \{a \in G.E \mid \exists e \in [a]_\mathsf{st}.\ \mathtt{lab}(e){=}(\mathtt{E}, -)\}$.

PSER Executions. An execution G is a *PSER execution* if: (1) events in $G.E$ have PSER labels; (2) each transaction class contains *exactly one* begin event; (3) each transaction class contains *at most one* end event; (4) each begin (resp. end) event is the first (resp. last) event (in po) within its transaction; and (5) only the last (po-maximal) transaction in each thread may be incomplete (due to a crash), i.e. $[E \setminus T]; \mathsf{po}_\mathsf{T}{=}\emptyset$. For simplicity, we further require that (6) transaction identifiers in each thread be ordered by po; that is, for all a, b: $(a,b) \in \mathsf{po}_\mathsf{T}|_{\mathsf{imm}} \iff \exists \tau, n.\ \mathtt{tx}(a){=}(\tau, n) \wedge \mathtt{tx}(b){=}(\tau, n{+}1)$.

Definition 11 (PSER-consistency)**.** A PSER execution $G{=}(E, I, P, \mathsf{po}, \mathsf{rf}, \mathsf{mo}, \mathsf{nvo})$ is *PSER-consistent* iff:

- $(\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb}) \cap G.\mathsf{st} \subseteq \mathsf{po}$ (SER1)
- hb_ser is irreflexive, where $\mathsf{hb}_\mathsf{ser} \triangleq (\mathsf{po}_\mathsf{T} \cup \mathsf{rf}_\mathsf{T} \cup \mathsf{mo}_\mathsf{T} \cup \mathsf{rb}_\mathsf{T})^+$ (SER2)
- $\mathsf{hb}_\mathsf{ser}|_D \subseteq \mathsf{nvo}$ (PSER-NVO)
- $\mathit{dom}([D]; G.\mathsf{st}; [P]) \subseteq P \subseteq G.T$ (PSER-ATOMIC1)
- $\mathsf{acyclic}(\mathsf{nvo}_\mathsf{T})$ (PSER-ATOMIC2)

The (SER1) and (SER2) axioms are those of serialisability [Papadimitriou 1979] adapted to our declarative framework as done e.g. in [Raad et al. 2018, 2019]. The (SER1) ensures that e.g. a

transaction observes its own writes by requiring $\mathsf{rf} \cap \mathsf{st} \subseteq \mathsf{po}$ (i.e. intra-transactional reads respect po). The (SER2) guarantees the existence of a total sequential order in which all concurrent transactions appear to execute atomically one after another. This total order is obtained by an arbitrary extension of the (partial) 'happens-before' relation $\mathsf{hb}_{\mathsf{ser}}$, which captures synchronisation resulting from transactional orderings imposed by program order (po_{T}) or conflict ($\mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}} \cup \mathsf{rb}_{\mathsf{T}}$).

Intuitively, $\mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}} \cup \mathsf{rb}_{\mathsf{T}}$ describes synchronisation due to conflicts between transactions. Two transactions are conflicted if they both access (read or write) a location x, and at least one of these accesses is a write. As such, the inclusion of $\mathsf{rf}_{\mathsf{T}} \cup \mathsf{mo}_{\mathsf{T}} \cup \mathsf{rb}_{\mathsf{T}}$ enforces conflict-freedom of serialisable transactions. For instance, if transactions ξ_1 and ξ_2 both write to x via events w_1 and w_2 such that $(w_1, w_2) \in \mathsf{mo}$, then ξ_1 must commit before ξ_2, and thus the entire effect of ξ_1 must be visible to ξ_2.

The (PSER-NVO), (PSER-ATOMIC1) and (PSER-ATOMIC2) axioms describe the persistency semantics of PSER. The (PSER-NVO) stipulates that transactional writes persist in the $\mathsf{hb}_{\mathsf{ser}}$ order. This in turn preserves inter-transactional synchronisation orderings across crashes. For instance, if ξ_2 reads from ξ_1, then ξ_1 persists before ξ_2; as such, upon recovery we never encounter the erroneous scenario where ξ_2 has persisted, whilst the transaction it read from, namely ξ_1, has not.

Lastly, (PSER-ATOMIC1) and (PSER-ATOMIC2) ensure that transactions *persist atomically*: (1) only complete transactions persist ($P \subseteq G.T$); (2) either all or none of the (durable) events in a transaction persist ($dom([D]; G.\mathsf{st}; [P]) \subseteq P$); and (3) the persists of a transaction are not interleaved by those of others: $\mathsf{acyclic}(\mathsf{nvo}_{\mathsf{T}})$.

Remark 3. Note that as all transactional writes are on persistent locations, we can require all or none of the events in a transaction to persist. Allowing transactional writes on volatile locations would violate persist atomicity as the writes on volatile locations do not survive a crash. Our notion of persist atomicity is inspired by the *write atomicity* of database transactions requiring that either all or none of the writes in a transaction commit. It is however straightforward to relax PSER to support mixed-volatility: we can require write atomicity on *all* writes during normal (non-crashing) executions, while requiring persist atomicity *only on writes of persistent locations* upon a crash.

5.1 PSER Utility: Persistently Linearisable Concurrent Library Implementations

Implementing and verifying concurrent libraries correctly is challenging. Typically, the library implementer is tasked with ensuring that the library state (e.g. a queue) remains *consistent* (e.g. a queue maintains its FIFO invariant), when simultaneously accessed by multiple threads. The library verifier is tasked with identifying the appropriate proof techniques to establish the desired consistency guarantees. One well-known such technique is the *linearisability* proofs of Herlihy and Wing [1990], which has been used extensively in the verification literature.

The challenges of implementing and verifying concurrent libraries are further compounded in the context of persistent hardware. Library implementers must additionally account for crashes and ensure that the library state remains both *consistent* and *persistent* across crashes. Library verifiers must accordingly adapt their proof techniques to establish the desired *consistency* and *persistency* guarantees. To do this, Izraelevitz et al. [2016b] introduced *buffered durable linearisability*, henceforth persistent linearisability, as an extension of linearisability in persistent settings.

We demonstrate that PSER can be used to streamline the tasks of implementing and verifying concurrent libraries in the persistent setting. In particular, we show PSER can convert *any* correct sequential implementation of a library into a correct (persistently linearisable) concurrent implementation. We proceed with the definitions of linearisability and persistent linearisability.

Linearisability and Persistent Linearisability. In linearisability proofs, a library call is typically represented as two *call events*, *inv* and *ack*, called a matching pair, denoting the call invocation and acknowledgement. To model executions of library clients, we define *library events* as the

extension of events (Def. 1) with *inv* and *ack* events. To identify each matching pair uniquely, we assume a finite set of *call identifiers*, CID, ranged over by ι. The labels of matching pairs are thus of the form $(\mathtt{I}, \iota, m, v_a)$ and $(\mathtt{A}, \iota, m, v_r)$, where m denotes the library method called, v_a denotes the invocation argument, and v_r denotes the return value.

A trace of a client program of a library is then represented as a *history* (a strict total order of events), to. As the library is concurrently accessed by multiple threads, the matching *inv* and *ack* events in a history may be interleaved with those of others. A history to is *sequential* if matching pairs in to are not interleaved by other call events in to. Given a set of events E and a relation r on E, a history to *linearises* $\langle E, \mathsf{r}\rangle$ if: (1) E can be extended to E_c by adding zero or more *ack* events; (2) E_c can be truncated to E_t by removing every *inv* in E_t without a matching *ack*; (3) to is a strict total order on E_t and includes r; and (4) to is a sequential history. The first step captures the notion that a pending *inv* may have taken effect even though its matching *ack* has not yet been returned; the second step captures the notion that the remaining pending invocations have not yet had an effect. An execution G of library $\mathcal{L}$ is *linearisable* if there exists a history to such that: (i) to linearises $\langle G.E, G.\mathsf{hb}\rangle$, where $G.\mathsf{hb} \triangleq (G.\mathsf{po} \cup G.\mathsf{rf})^+$ denotes the 'happens-before' relation; and (ii) to is a *legal* history. The definition of legal histories is library-specific, e.g. the FIFO property of queue histories. A program P is *linearisable* if all its consistent executions are linearisable.

Persistent linearisability extends linearisability to persistent settings. Note that given a chain $C{=}G_1, \cdots, G_n$, due to asynchronicity of persists in buffered models, in each era G_i (except G_n) *only a subset* of events in $G.E$ may persist prior to a crash, i.e. those in $G.P$. As such, a chain $C{=}G_1, \cdots, G_n$ is *persistently linearisable* if there exist $\mathsf{to}_1 \cdots \mathsf{to}_n$ such that: (i) each to_i linearises $\langle G.P, G.\mathsf{hb}\rangle$; and (ii) $\mathsf{to}_1 +\!\!+ \cdots +\!\!+ \mathsf{to}_n$ is a legal history, where $+\!\!+$ denotes sequence concatenation. A persistent program $\mathbb{P}$ is *persistently linearisable* if all its valid chains are persistently linearisable.

Definition 12 (Library events)**.** Assume a finite set of *call identifiers*, CID, ranged over by ι. Given a library $\mathcal{L}$ and its associated set of *operations* $\textsc{Op}_{\mathcal{L}} \subseteq \textsc{String}$, A *library event* of $\mathcal{L}$ is a tuple $\langle n, \tau, l\rangle$, where $n \in \mathbb{N}$ is an event identifier, $\tau \in \textsc{TID}$ is a thread identifier, and $l \in \textsc{Lab} \cup \left\{(\mathtt{I}, \iota, m, v_a), (\mathtt{A}, \iota, m, v_r) \;\middle|\; \iota \in \textsc{CID} \wedge m \in \textsc{Op}_{\mathcal{L}} \wedge v_a, v_r \in \textsc{Val}\right\}$ is an event label with LAB as described in Def. 1. The set of *invocation* events is : $I \triangleq \left\{e \;\middle|\; \mathtt{lab}(e){=}(\mathtt{I}, -, -, -)\right\}$. The set of acknowledgement events, A, is defined analogously. The set of *matching call pairs* is: $\mathsf{Match} \triangleq \left\{(e_i, e_a) \;\middle|\; \exists \iota, m.\ \mathtt{lab}(e_i){=}(\mathtt{I}, \iota, m, -) \wedge \mathtt{lab}(e_a){=}(\mathtt{A}, \iota, m, -)\right\}$.

The definitions of execution graphs (Def. 4) and execution chains (Def. 5) are simply lifted to admit library events. As such, we write e.g. *library execution graph* for an execution graph whose events are library events. Given a library execution graph G, we assume that call identifiers are unique across matching pairs in $G.E$, i.e. no two *inv* (resp. *ack*) events have the same call identifier.

Definition 13 (Persistent linearisability)**.** Given a set of events E and a relation $\mathsf{r} \subseteq E \times E$, a history (strict total order of events) to *linearises* $\langle E, \mathsf{r}\rangle$ iff there exist E_c and E_t such that:

- $E_c \in \mathsf{comp}(E)$ with $\mathsf{comp}(S) \triangleq \left\{ S' \supseteq S \;\middle|\; \begin{array}{l} S' \setminus S \subseteq A \wedge \forall e_a \in S' \setminus S. \\ \quad \exists e_i \in S.\ (e_i, e_a) \in \mathsf{Match} \\ \quad \wedge\ \nexists e'_a \in S.\ (e_i, e'_a) \in \mathsf{Match} \\ \quad \wedge\ \forall e'_a \in S' \cap A.\ \mathtt{cid}(e_a){=}\mathtt{cid}(e'_a) \Rightarrow e_a{=}e'_a \end{array} \right\}$;
- $E_t = \mathtt{trunc}(E_c)$ with $\mathtt{trunc}(S) \triangleq (I \cup A) \cap \left(S \setminus \left\{i \in I \;\middle|\; \nexists a \in S.\ (i, a) \in \mathsf{Match}\right\}\right)$;
- to is an enumeration of E_t such that: $\forall a, b \in E_t.\ (a, b) \in \mathsf{r} \Rightarrow (a, b) \in \mathsf{to}$; and
- to is *sequential*, i.e. is of the form $i_1; a_1; \cdots; i_m; a_m$, with each $(i_k, a_k) \in \mathsf{Match}$.

An Execution G of library $\mathcal{L}$ is *linearisable* iff there exist to such that: (1) to linearises $\langle G.P, G.\mathsf{hb}\rangle$, where $G.\mathsf{hb} \triangleq (G.\mathsf{po} \cup G.\mathsf{rf})^+$; and (2) to is a legal history of $\mathcal{L}$. A program P is *linearisable* if all its consistent executions are linearisable.

A chain $C{=}G_1, \cdots, G_n$ of library $\mathcal{L}$ is *persistently linearisable* iff there exist $\mathsf{to}_1 \cdots \mathsf{to}_n$ such that: (1) each to_i linearises $\langle G_i.P, G_i.\mathsf{hb}\rangle$; and (2) $\mathsf{to}_1 +\!\!+ \cdots +\!\!+ \mathsf{to}_n$ is a legal history of $\mathcal{L}$. A persistent program $\mathbb{P}$ is *persistently linearisable* if all its valid chains are persistently linearisable.

PSER for Concurrent Library Implementation. We next show that given a library $\mathcal{L}$, and *any* correct sequential implementation $\mathcal{I}$ of $\mathcal{L}$, we can use PSER to convert $\mathcal{I}$ into a correct, concurrent and persistent implementation as follows. Consider a library $\mathcal{L}$ with operations $m_1 \cdots m_n$, e.g. a queue library with enq and deq operations. Given a sequential implementation $\mathcal{I}$ of $\mathcal{L}$, let $\mathcal{I}(m_i)$ denote the implementation (body) of m_i in $\mathcal{I}$, for each m_i. We can then convert $\mathcal{I}$ into a concurrent implementation, $\mathcal{I}_{\mathsf{tx}}$, by wrapping each $\mathcal{I}(m_i)$ in a PSER transaction. If the sequential implementation $\mathcal{I}$ is correct, then the concurrent implementation $\mathcal{I}_{\mathsf{tx}}$ is *persistently linearisable*.

Note that $\mathcal{I}_{\mathsf{tx}}$ is persistently linearisable only if the sequential implementation $\mathcal{I}$ is *correct*. For instance, a bogus queue implementation where deq always returns value 7 is not correct and thus cannot be converted into a persistently linearisable concurrent implementation. To rule out such erroneous implementations, we require that $\mathcal{I}$ be *sequentially sound*. An implementation $\mathcal{I}$ of library $\mathcal{L}$ is sequentially sound if its sequential execution always yields a legal history.

Definition 14. An implementation $\mathcal{I}$ is *sequentially sound* iff for all programs P, executions G of P and sequential histories to of $G.E$, if $G.\mathsf{hb} \subseteq \mathsf{to}$, then to is a legal history.

Given a client program P, let P_{tx} denote the program obtained from P by replacing every call to $\mathcal{I}$ with a call to $\mathcal{I}_{\mathsf{tx}}$. To show that $\mathcal{I}_{\mathsf{tx}}$ is persistently linearisable, we must show that given an arbitrary PSER program P, all valid *chains* of P_{tx} are persistently linearisable. However, recall that chains are associated with *persistent programs* (Def. 6). We thus define the *PSER recovery mechanism*, $\mathtt{rec}_{\mathrm{PSER}}$, for recovering from a crash under PSER. Given an execution G of a PSER program P, $\mathtt{rec}_{\mathrm{PSER}}(\mathsf{P}, G)$ resumes P from the last persisted transaction in each thread. That is, given that under PSER the transactions persist atomically and in the po order of each thread, $\mathtt{rec}_{\mathrm{PSER}}(\mathsf{P}, G)$ identifies the latest (in po) persisted transaction in each thread and resumes execution thereafter. Recall that given a PSER program $\mathsf{P} \in \textsc{Prog}_{\mathrm{PSER}}$, the sequential program in each thread τ is a sequence of transactions: $\mathsf{P}[\tau]{=}[\mathsf{C}_1]; \cdots; [\mathsf{C}_m]$; we thus write $\mathsf{sub}(\mathsf{P}[\tau], k)$ to denote the sub-program $[\mathsf{C}_k]; \cdots; [\mathsf{C}_m]$.

Definition 15 (PSER recovery)**.** The *PSER recovery mechanism* is: $\mathtt{rec}_{\mathrm{PSER}}(\mathsf{P}, G) \triangleq \lambda\tau.\mathsf{sub}(\mathsf{P}[\tau], n{+}1)$, with $n{=}\max\left(\{i \mid \exists e{\in}G.T.\, \mathsf{tx}(e){=}(\tau, i) \wedge \mathit{dom}([D]; \mathrm{po}_{\mathsf{T}}{}^{?}; [e]_{\mathrm{st}}) \subseteq P\}\right)$.

Theorem 1 (Linearisability)**.** *Given an implementation $\mathcal{I}$ of library $\mathcal{L}$, if $\mathcal{I}$ is sequentially sound, then for all PSER programs* P*: (1)* P_{tx} *is linearisable; and (2)* $\langle \mathsf{P}_{\mathsf{tx}}, \mathtt{rec}_{PSER}\rangle$ *is persistently linearisable.*

Proof. The full proof is given in the accompanying technical appendix.

6 A PSER IMPLEMENTATION IN PARMv8

We develop a sound implementation of PSER and its recovery mechanism in PARMv8, thus demonstrating the feasibility of our PSER model through correct compilation to PARMv8.

Notation. We shortly present our PSER implementation in Fig. 4. *All* memory locations accessed in the implementation are persistent; we thus write e.g. x rather than x_p for better readability. Recall that for brevity we omit the access modes when relaxed, and write e.g. $x{:=}\,\mathsf{b}$ for $x{:=}_{\mathtt{rlx}}\,\mathsf{b}$. As we often need to persist writes using write-backs, we write $x{:=}_{\mathtt{wb}}\,\mathsf{e}$ as a shorthand for $x{:=}\,\mathsf{e}$; **wb** x.

MRSW Locks. As we describe shortly, our PSER implementation in Fig. 4 uses *locks* to synchronise concurrent accesses to shared data. As serialisability allows concurrent transactions to read from the same memory location simultaneously, for better performance we use MRSW (multiple-readers-single-writer) locks. We thus assume that each location x is associated with an MRSW

```
0. [C]PSER→PARMv8 ≜
1.  LS := ∅;
2.  RS := ∅; WS := ∅;
3.  τ := getTID(); ξ := getTxID();
4.  log[τ] :=wb ξ;  w :=wb new-array();
5.  ⟪C⟫; DSBfull;
6.  ws[ξ] :=wb w;
7.  for (x ∈ WS) {
8.    if (promote(x)) LS.add(x);
9.    else {
10.     for (x ∈ LS) w-unlock(x);
11.     for (x ∈ (WS∪RS)\LS) r-unlock(x);
12.     goto line 1; } }
13.  for (x ∈ WS) {
14.    a := w[x];
15.    x :=wb a;
16.  }
17.  DSBfull;
18.  for (x ∈ WS) w-unlock(x);
19.  for (x ∈ RS\WS) r-unlock(x);
```

```
⟪x := a⟫ ≜ if (x ∉ RS ∪ WS) {
             r-lock(x);
             l[x] :=wb ξ;
           } WS.add(x);
           w[x] :=wb a;

⟪a := x⟫ ≜ if (x ∉ RS ∪ WS) {
             r-lock(x);
             l[x] :=wb ξ;
           } RS.add(x);
           if (x ∉ WS)
             a := x;
           else
             a := w[x];

⟪C1 ; C2⟫ ≜ ⟪C1⟫ ; ⟪C2⟫

        ...
```

```
20. recover(P) ≜
21.  for (x ∈ dom(l))
22.    w-unlock(x);
23.  for (τ ∈ dom(P)) {
24.    ξ := log[τ];
25.    w := ws[ξ];
26.    if (w = ⊥)
27.      P'[τ] := sub(P[τ], ξ);
28.    else {
29.      P'[τ] := sub(P[τ], ξ + 1);
30.      if (!committed(w, ξ)) {
31.        for (x ∈ dom(w))
32.          x :=wb w [x];
33.      }
34.    }
35.  }
36.  DSBfull;
37.  run(P');
```

where committed(w, ξ) $\overset{\text{def}}{\iff}$ $dom(w)=\emptyset \vee \exists x, \xi'.\ x \in dom(w) \wedge \xi' \neq \xi \wedge l[x]=\xi'$

Fig. 4. PSER implementation of transaction [C] in PARMv8 (left ¦ middle) where the grey code ensures deadlock avoidance and the highlighted code ensures persistency; PSER recovery implementation in PARMv8 (right).

lock which can be acquired by either (i) multiple threads reading from x simultaneously; or (ii) a single thread writing to x. A reader (resp. writer) lock on x is acquired by calling r-lock(x) (resp. w-lock(x)), and released by calling r-unlock(x) (resp. w-unlock(x)). Moreover, a reader lock on x can be *promoted* to a writer one by calling promote(x). As two distinct reader locks on x may simultaneously attempt to promote their locks, promotion is done on a 'first-come-first-served' basis. A call to promote(x) thus returns a boolean denoting either (i) successful promotion (true); or (ii) failed promotion as another reader lock on x is currently being promoted (false). A call to promote(x) returns successfully once all other readers have released their locks on x and thus the calling reader can safely assume exclusive ownership of the lock (in write mode). Our MRSW lock implementation is straightforward, and is provided in the accompanying technical appendix.

Serialisability of Our PSER Implementation. Given a transaction [C], our PSER implementation of C in PARMv8, written $[C]_{\text{PSER}\to\text{PARMv8}}$, is given in Fig. 4 (left). Ignoring the code in grey (lines 1, 8–12), and the highlighted code, $[C]_{\text{PSER}\to\text{PARMv8}}$ describes a serialisable implementation of C using MRSW locks. Let RS and WS respectively denote the *read set* and *write set* of C, i.e. the locations read and written by C. Conceptually, a serialisable implementation of C would: (i) acquire the locks on all locations in RS ∪ WS; (ii) execute C *locally* where the reads in C are carried out in place (read directly from memory), while the writes are recorded tentatively in a log w; (iii) commit the *effect* of C (in w) by propagating the writes in w to memory; and (iv) release the acquired locks.

Note that the locations accessed by a transaction are not known in advance; i.e. the RS and WS are not known beforehand. As such, we cannot acquire all necessary locks at the beginning as stated in step (i) above. Instead, we compute RS and WS incrementally, acquiring the necessary locks *on the fly*, by combining steps (i)-(ii) above. Moreover, to reduce lock contention as much as possible, we acquire all necessary locks in read mode, and promote the locks on WS just before committing. Our serialisable implementation thus proceeds as follows. Starting with empty RS and WS (line 2), and an empty write log w (line 4), we execute C locally (as described above) whilst acquiring the

necessary locks on the fly. This is denoted by $(\!|\mathsf{C}|\!)$ on line 5, as described shortly. Once the local execution $(\!|\mathsf{C}|\!)$ is completed, we promote the locks on WS (lines 7–8), commit the writes recorded in w to memory (lines 13–15), and finally release all acquired locks (lines 18–19).

The local execution $(\!|\mathsf{C}|\!)$ is given in Fig. 4 (middle), and is obtained from C as follows. For each write operation $x := \mathsf{a}$, the WS is extended with x, and the written value is logged in $w[x]$. Recall that to reduce lock contention, for each written location x, our implementation first acquires a reader lock on x, and subsequently promotes it to a writer lock. As such, the local execution of $x := \mathsf{a}$ first checks if a reader lock for x has been acquired (i.e. $x \in \mathsf{RS} \cup \mathsf{WS}$) and obtains one if this is not the case. Analogously, for each read operation $\mathsf{a} := x$, a reader lock is acquired if necessary and RS is extended with x. Moreover, as each transaction must observe its own writes, the local execution of $\mathsf{a} := x$ first checks if x has been written to by itself (i.e. $x \in \mathsf{WS}$). If this is not the case the value of x is read from the memory; otherwise, the value of x is read from the log w. The local execution of the remaining inductive cases (e.g. $\mathsf{C}_1; \mathsf{C}_2$) is defined by straightforward induction on the structure of commands (e.g. $(\!|\mathsf{C}_1; \mathsf{C}_2|\!) \triangleq (\!|\mathsf{C}_1|\!); (\!|\mathsf{C}_2|\!)$), and is omitted here.

Avoiding Deadlocks. Recall that a call to promote(x) by reader r returns false when another reader r' is in the process of promoting a lock on x. When this is the case, r must release its reader lock on x to ensure the successful promotion of x by r' and thus avoid deadlocks. To this end, our implementation includes a deadlock avoidance mechanism (lines 8–12) as follows. We record a set LS (initialised with $\emptyset$ on line 1) of those locks on the write set that have been successfully promoted so far. When promoting a lock on x succeeds (line 8), then LS is extended with x. On the other hand, when promoting x fails (line 9), all those locks promoted so far (i.e. in LS) as well as the other reader locks acquired thus far (i.e. in $\mathsf{WS} \cup \mathsf{RS} \setminus \mathsf{LS}$) are released and the transaction is restarted.

Persistency of PSER Implementation. Recall that given a PSER program $\mathsf{P} \in \textsc{Prog}_{\textsc{pser}}$, we assume that the sequential program in each thread $\tau_i \in dom(\mathsf{P})$ comprises a sequence of transactions, i.e. $\mathsf{P}(\tau_i) = [\mathsf{C}_i^1]; \cdots; [\mathsf{C}_i^n]$. We thus represent $\mathsf{P}(\tau_i)$ as an array C_i such that $C_i[j] = [\mathsf{C}_i^j]$. We further assume that the context of each thread τ_i is set up such that: (1) a call to getTID() returns i; and (2) a call to getTxID() returns j when executing $[\mathsf{C}_i^j]$. A program P is executed by calling run(P).

To ensure correct recovery, our implementation must account for the possibility of a crash at each program point. To do this, we record the metadata for tracking the progress of each thread in *log*, *ws* and *l*, as follows. For each thread τ, $log[\tau]$ records the last executed transaction; for each transaction ξ, $ws[\xi]$ records the effect of ξ; and for each location x, $l[x]$ records the last transaction that acquired a lock on x. As such, when thread τ executes transaction ξ (line 3) with transaction code given by C, our implementation logs ξ in $log[\tau]$ (line 4); records the transaction's effect in $ws[\xi]$ (line 6); and records ξ in $l[x]$ for each location x accessed in C (via $(\!|\mathsf{C}|\!)$ on line 5).

Recall that the transaction effect is computed incrementally in w via the local execution $(\!|\mathsf{C}|\!)$. For correct recovery, we must ensure that the transaction effect is persisted *fully* and *not partially* in case of a crash. To achieve this, before recording the effect w in $ws[\xi]$ on line 6, we insert a $\mathbf{DSB}_{\mathtt{full}}$ instruction (line 5) to ensure that all pending writes, including those of w, are persisted to memory.

Observe that our implementation adheres to the following pattern: (1) it updates the metadata for tracking the thread progress (lines 3–4); (2) executes a $\mathbf{DSB}_{\mathtt{full}}$ (line 5); (3) executes the transaction (lines 7–15); and (4) executes a $\mathbf{DSB}_{\mathtt{full}}$ (line 17). The first two steps ensure that the recovery metadata of each thread does not lag behind its progress; conversely, the last two steps ensure that the progress of each thread does not lag behind its recovery metadata. Therefore, in case of a crash, the persisted progress of each thread τ may at most be one step behind its persisted metadata.

PSER Recovery Implementation. After a crash, a program P is restored by calling `recover(P)` in Fig. 4 (right), which releases all locks to avoid deadlocks (lines 21–22); restores the progress of threads by generating a new program P' (lines 23–36); and ultimately runs P' (line 37).

Recall that the persisted progress of each thread is at most one step behind its persisted metadata. As such, it suffices to check whether the effect of the *last* recorded transaction for τ has persisted, and to resume the execution of τ accordingly. More concretely, let the last transaction executed by τ be ξ (line 24) and let us read the effect of ξ in the local variable w (line 25). Then, either (i) the effect has not persisted before the crash (i.e. the crash occurred before line 6) and thus $w{=}\bot$ and $P[\tau]$ is resumed from ξ (line 27), or (ii) the effect has persisted (i.e. the crash occurred after line 6) and thus $P[\tau]$ is advanced to $\xi{+}1$ (line 29), where $\mathsf{sub}(P[\tau], n)$ denotes the subarray of $P[\tau]$ at n.

Note that in case (ii), the effect of ξ (in w) may not have fully committed or persisted to memory (e.g. if the crash occurred before line 13), and we must thus commit the transaction effect (lines 31–35). This is ascertained via `committed(w, ξ)` on line 30, checking if the writes of ξ in w have fully persisted. The `committed(w, ξ)` predicate is defined in Fig. 4. When $dom(w){=}\emptyset$, the transaction is read-only and w is vacuously persisted. When $dom(w){\neq}\emptyset$ and $x \in dom(w)$, we can safely assume w has persisted if another transaction $\xi'{\neq}\xi$ is the *last* transaction to acquire the lock on x (i.e. $l[x]{=}\xi'$). More concretely, since w has persisted, the crash must have occurred after line 6. That is, the $(\!|C|\!)$ on line 5 has fully persisted and thus the lock on x was acquired by ξ (as $x \in dom(w)$). Consequently, as ξ' is the last transaction to acquire the x lock, then ξ must have released the lock on x (line 18), i.e. ξ has fully committed and persisted. Finally, the $\mathbf{DSB}_{\mathtt{full}}$ on line 36 ensures that the committed writes are persisted before restarting P'.

Theorem 2 (Soundness). *The PSER implementation and its recovery mechanism in Fig. 4 are sound.*

Proof. The full proof is given in the accompanying technical appendix.

7 CONCLUSIONS AND FUTURE WORK

Although research into burgeoning NVM technologies has grown rapidly over the recent years, the *formal* study of NVM persistency semantics has remained largely unexplored, both at the architecture and the language levels. To close this gap, we developed a formal declarative framework for describing concurrency models in the NVM context. We then presented the PARMv8 model as an instance of this framework, formalising the ARM persistency semantics for the first time. To streamline NVM programming, we developed the PSER model (as another instance of our framework) as the first formal transactional model in the NVM context. We then showed the PSER utility for correct, concurrent and persistent library implementations. Finally, we developed a sound implementation of PSER in PARMv8, demonstrating correct PSER-to-PARMv8 compilation.

As discussed in §2.2, we have encoded our PARMv8 and PSER models in Alloy [Jackson 2012], and provide our model files in our supplementary material. However, we have not attempted to use our models to generate conformance tests, due to the complications introduced by the nvo relation. While for consistency models it is well-known how to write a test for a desired mo relation (e.g. by adding a concurrent thread with a sequence of loads), we cannot inspect the nvo relation in the same way. More concretely, the only reliable way to ensure that we observe persistent data is to contrive a crash and read the data afterwards, since post-crash reads retrieve persistent data. However, this only allows us to observe the nvo-*latest* write on each location before the crash, and is not sufficient for inferring the nvo order on the other writes. For instance, when $P \triangleq x{:=}1 || \cdots || x{:=}n$ crashes and $x{=}i$ afterwards, we can deduce that $x{:=}i$ was the nvo-maximal write that persisted, but cannot infer the nvo order between the other $n{-}1$ writes.

More generally, experimental validation of memory persistency models remains an interesting open problem, and we plan to pursue it in future work. Litmus testing for memory *consistency*

models is well established [Alglave et al. 2015, 2011; Chong et al. 2018], but it remains unclear how best to insert the crashes that would be needed to test memory *persistency* models. One option would be to use a simulator (which is how Bornholt et al. [2016] validated their crash-consistency models for filesystems) but this would not necessary reveal all the concurrency behaviours that real processors can exhibit. Another option, applicable when the processor-under-test is a component of a system-on-chip (SoC) FPGA [Jain et al. 2018], is to build custom hardware to monitor the traffic to persistent memory, and thus to detect nvo violations without the need for crashes.

As additional directions of future work, we plan to build on top of the work presented here in several ways. First, having established a sound baseline PSER implementation in PARMv8, we plan to investigate its performance, and the extent to which its performance can be improved by the selective removal of barriers and write-backs [Alglave et al. 2017]. Second, we plan to explore language-level persistency models further by (1) investigating weaker transactional consistency models than serialisability (e.g. snapshot isolation) in the context of NVM; and (2) researching persistency extensions of high-level languages such as C/C++. Third, we plan to formalise other architecture-level persistency models, including that of Intel-x86 described informally in [Intel 2019]; such a formalisation would provide a firmer foundation for tools such as PMTest [Liu et al. 2019] that seek persistency bugs in software running over x86. Fourth, we plan to specify and verify the persistency semantics and guarantees of existing NVM libraries, such as those in [Cooper 2008; Intel 2015; PCJ 2016]. Lastly, using our formal declarative framework, we plan to develop automated verification techniques for NVM, such as model checking.

ACKNOWLEDGMENTS

We thank the OOPSLA'19 reviewers for their valuable feedback. We thank William Wang for his insightful feedback and helpful discussions. The first author was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", under the European Union Horizon 2020 Framework Programme (grant agreement number 683289). The second author was supported in part by the EPSRC grant EP/R006865/1.

REFERENCES

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. 2015. GPU concurrency: Weak behaviours and programming assumptions. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2015)*. ACM, New York, NY, USA, 577–591. https://doi.org/10.1145/2694344.2694391

Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 6 (May 2017), 38 pages. https://doi.org/10.1145/2994593

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running tests against hardware. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2011)*, Vol. 6605. Springer, 41–44. https://doi.org/10.1007/978-3-642-19835-9_5

ARM. 2018. ARM architecture reference manual ARMv8, for ARMv8-A architecture profile (DDI 0487D.a). https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf

Hillel Avni, Eliezer Levy, and Avi Mendelson. 2015. Hardware transactions in nonvolatile memory. In *29th International Symposium on Distributed Computing (DISC 2015)*. Springer, 617–630. https://doi.org/10.1007/978-3-662-48653-5_41

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence programming models for non-volatile memory. In *2016 ACM SIGPLAN International Symposium on Memory Management (ISMM 2016)*. ACM, New York, NY, USA, 55–67. https://doi.org/10.1145/2926697.2926704

James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and checking file system crash-consistency models. In *21st International Conference on Architectural Support for Programming*

Languages and Operating Systems (ASPLOS 2016). ACM, 83–98. https://doi.org/10.1145/2872362.2872406

Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *SIGPLAN Not.* 49, 10 (Oct. 2014), 433–452. https://doi.org/10.1145/2714064.2660224

Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. https://doi.org/10.14778/2735479.2735483

Nathan Chong, Tyler Sorensen, and John Wickerson. 2018. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 211–225. https://doi.org/10.1145/3192366.3192373

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. *SIGPLAN Not.* 46, 3 (March 2011), 105–118. https://doi.org/10.1145/1961296.1950380

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 *ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP 2009)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

Harold Cooper. 2008. Persistent collections. https://pcollections.org/

Niall Douglas. 2018. P1026R0: A call for a data persistence (iostream v2) study group. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1026r0.pdf

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News* 18, 2SI (May 1990), 15–26. https://doi.org/10.1145/325096.325102

Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions. In *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

Intel. 2014. Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf

Intel. 2015. Persistent memory programming. http://pmem.io/

Intel. 2019. Intel 64 and IA-32 architectures software developer's manual (Combined volumes). https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

ITRS. 2011. Process integration, devices, and structures. http://www.maltiel-consulting.com/ITRS_2011-Process-Integration-Devices-Structures.pdf International technology roadmap for semiconductors.

Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016a. Failure-atomic persistent memory updates via JUSTDO logging. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016b. Linearizability of persistent memory objects under a full-system-crash failure model. In *30th International Symposium on Distributed Computing (DISC 2016)*. Springer, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

Daniel Jackson. 2012. *Software abstractions – Logic, language, and analysis* (revised ed.). MIT Press.

Abhishek Kumar Jain, G. Scott Lloyd, and Maya Gokhale. 2018. Microscope on memory: MPSoC-enabled computer memory system assessments. In *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM 2018)*. IEEE, 173–180. https://doi.org/10.1109/FCCM.2018.00035

Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2830772.2830805

T. Kawahara, K. Ito, R. Takemura, and H. Ohno. 2012. Spin-transfer torque RAM technology: Review and prospect. *Microelectronics Reliability* 52, 4 (2012), 613 – 627. https://doi.org/10.1016/j.microrel.2011.09.028

Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level persistency. In *44th Annual International Symposium on Computer Architecture (ISCA 2017)*. ACM, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016a. High-performance transactions for persistent memories. *SIGPLAN Not.* 51, 4 (March 2016), 399–411. https://doi.org/10.1145/2954679.2872381

Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016b. Delegated persist ordering. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 58, 13 pages. https://doi.org/10.1109/MICRO.2016.7783761

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM,

New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *36th Annual International Symposium on Computer Architecture (ISCA 2009)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Manabi Khan. 2019. PMTest: A fast and flexible testing framework for persistent memory programs. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*. ACM, 411–425. https://doi.org/10.1145/3297858.3304015

Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated synthesis of comprehensive memory model litmus test suites. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017)*. ACM, 661–675. https://doi.org/10.1145/3037697.3037723

Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017) (LIPIcs)*, Vol. 91. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 37:1–37:16. https://doi.org/10.4230/LIPIcs.DISC.2017.37

Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J. ACM* 26, 4 (Oct. 1979), 631–653. https://doi.org/10.1145/322154.322158

PCJ. 2016. Persistent collections for Java. https://github.com/pmem/pcj

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

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (Dec. 2018), 29 pages. https://doi.org/10.1145/3158107

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2018. On parallel snapshot isolation and release/acquire consistency. In *Programming Languages and Systems (ESOP 2018)*. Springer, 940–967. https://doi.org/10.1007/978-3-319-89884-1_33

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2019. On the semantics of snapshot isolation. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2019)*. Springer, 1–23. https://doi.org/10.1007/978-3-030-11245-5_1

Azalea Raad and Viktor Vafeiadis. 2018. Persistence semantics for weak memory: Integrating epoch persistency with the TSO memory model. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 137 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276507

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (July 2010), 89–97. https://doi.org/10.1145/1785414.1785443

Hongping Shu, Hongyu Chen, Hao Liu, Youyou Lu, Qingda Hu, and Jiwu Shu. 2018. Empirical study of transactional management for persistent memory. In *7th Non-Volatile Memory Systems and Applications Symposium (NVMSA 2018)*. IEEE, 61–66. https://doi.org/10.1109/NVMSA.2018.00015

D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. 2008. The missing memristor found. *Nature* 453 (2008), 80 – 83.

Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, Ankit Singla, Pratap Subrahmanyam, and Onur Mutlu. 2018. Enabling efficient RDMA-based synchronous mirroring of persistent memory transactions. *CoRR* abs/1810.09360 (2018). arXiv:1810.09360

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not.* 47, 4 (March 2011), 91–104. https://doi.org/10.1145/2248487.1950379

John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. 2017. Automatically comparing memory consistency models. In *44th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. 190–204. https://doi.org/10.1145/3009837.3009838

Xiaojian Wu and A. L. Narasimha Reddy. 2011. SCMFS: A file system for storage class memory. In *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2011)*. ACM, New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2063384.2063436

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 *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744