MEMENTO: A Framework for Detectable Recoverability in Persistent Memory

KYEONGMIN CHO, KAIST, Korea
SEUNGMIN JEON, KAIST, Korea
AZALEA RAAD, Imperial College London, United Kingdom
JEEHOON KANG, KAIST, Korea

Persistent memory (PM) is an emerging class of storage technology that combines the performance of DRAM with the durability of SSD, offering the best of both worlds. This had led to a surge of research on persistent objects in PM. Among such persistent objects, concurrent data structures (DSs) are particularly interesting thanks to their performance and scalability. One of the most widely used correctness criteria for persistent concurrent DSs is *detectable recoverability*, ensuring both thread safety (for correctness in non-crashing concurrent executions) and crash consistency (for correctness in crashing executions). However, the existing approaches to designing detectably recoverable concurrent DSs are either limited to simple algorithms or suffer from high runtime overheads.

We present MEMENTO: a *general* and *high-performance* programming framework for detectably recoverable concurrent DSs in PM. To ensure general applicability to various DSs, MEMENTO supports primitive operations such as checkpoint and compare-and-swap and their composition with control constructs. To ensure high performance, MEMENTO employs a timestamp-based recovery strategy that requires fewer writes and flushes to PM than the existing approaches. We formally prove that MEMENTO ensures detectable recoverability in the presence of crashes. To showcase MEMENTO, we implement a lock-free stack, list, queue, and hash table, and a combining queue that detectably recovers from random crashes in stress tests and performs comparably to existing hand-tuned persistent DSs with and without detectable recoverability.

CCS Concepts: • **Theory of computation → Concurrent algorithms**; • **Computer systems organization → Reliability**; • **Hardware → Non-volatile memory**.

Additional Key Words and Phrases: persistent memory, detectable recovery, concurrent data structure

ACM Reference Format:
Kyeongmin Cho, Seungmin Jeon, Azalea Raad, and Jeehoon Kang. 2023. MEMENTO: A Framework for Detectable Recoverability in Persistent Memory. *Proc. ACM Program. Lang.* 7, PLDI, Article 118 (June 2023), 26 pages. https://doi.org/10.1145/3591232

1 INTRODUCTION

Persistent memory (PM) technologies such as Samsung's recently announced Memory-Semantic SSD [Samsung 2023] and Kioxia's XL-FLASH [Choe 2022] simultaneously provide **(1)** low-latency, high-throughput, and fine-grained data transfer capabilities as DRAM does; and **(2)** durable and high-capacity storage as SSD does. As such, PM has the potential to radically change the way we build fault-tolerant systems by optimizing traditional and distributed file systems [Chen et al. 2021; Kadekodi et al. 2021; Kim et al. 2021a; Kwon et al. 2017; Xu and Swanson 2016; Zhu et al. 2021],

Authors' addresses: Kyeongmin Cho, KAIST, Daejeon, Korea, kyeongmin.cho@kaist.ac.kr; Seungmin Jeon, KAIST, Daejeon, Korea, seungmin.jeon@kaist.ac.kr; Azalea Raad, Imperial College London, London, United Kingdom, azalea.raad@imperial.ac.uk; Jeehoon Kang, jeehoon.kang@kaist.ac.kr, KAIST, Daejeon, Korea.

2475-1421/2023/6-ART118
https://doi.org/10.1145/3591232

transaction processing systems for high-velocity real-time data [Meehan et al. 2015], distributed stream processing systems [Wang et al. 2021], and stateful applications organized as a pipeline of cloud serverless functions interacting with cloud storage systems [Setty et al. 2016; Zhang et al. 2020].

A key building block in PM for such optimizations are concurrent data structures (DSs) that ensure that the underlying DS is both *thread-safe* (i.e. it behaves correctly when accessed by concurrent threads racing to manipulate the DS) and *crash-consistent* (i.e. it is restored into a consistent state upon recovery from a crash, e.g. a power failure). The thread-safety of the underlying DS is ensured by using a suitable *concurrency control* mechanism, e.g. transactional memory (TM), locking, or lock-free techniques using fine-grained synchronization primitives (e.g. CAS instructions). Compared to TM- or locking-based DS implementations, *lock-free* data structures have the following two advantages. **(1)** They have a greater potential to parallelize workloads than others by distributing memory accesses across a multitude of contention points [David et al. 2018]. For instance, logging imposes significant overhead both in time (due to the concentrated contention point at the tip) and in space (because all intermediate changes are recorded). As such, lock-free queues and hash tables [Fatourou and Kallimanis 2011, 2012; Goodman et al. 1989; Hendler et al. 2010] outperform lock- and TM-based ones in PM. **(2)** Lock-free algorithms ensure that the DS is in a consistent state *at all times*, thereby eliminating the need for additional mechanisms to ensure crash consistency, so long as the updates on the DS are flushed to PM in a timely manner.

As such, persistent lock-free DSs have drawn significant attention in the literature, including persistent lock-free stacks [Attiya et al. 2019], queues [Friedman et al. 2018], lists [Attiya et al. 2022; Zuriel et al. 2019], hash tables [Chen et al. 2020; Nam et al. 2019; Zuriel et al. 2019], and trees [Attiya et al. 2022], as well as general techniques for transforming volatile (in-DRAM) lock-free DSs to persistent (in-PM) DSs [Friedman et al. 2020, 2021; Izraelevitz et al. 2016; Lee et al. 2019].

One of the most widely accepted correctness criteria for persistent lock-free DSs (and concurrent DSs in general) is *durable linearizability* (DL) [Izraelevitz et al. 2016]. A multi-threaded execution (where the operations of concurrent threads can arbitrarily interleave) is *linearizable* if each operation appears to execute and take effect atomically (without being interleaved by operations in other threads) at some point, called the *linearization point*, between its invocation and response [Herlihy and Wing 1990]. DL is an extension of linearizability to the PM setting and additionally offers crash consistency guarantees. Specifically, a multi-threaded execution that possibly spans multiple crashes satisfies DL if it is linearizable when ignoring the crash events. In particular, operations completed before a crash should be persisted across the crash, and if there are operations whose executions are interrupted by the crash, then the DS should be *recovered* to a consistent state after the crash. DL is indeed satisfied by most existing persistent concurrent DSs, except for those DSs that intentionally trade durability for performance [Friedman et al. 2018].

However, DL is insufficient for *composing* persistent DSs with one another [Friedman et al. 2018]. For instance, consider a banking DS comprising a *savings* account, S, and a *current* account, C, where amount a is withdrawn from S and, if successful, deposited into C:

```
1: $succ \leftarrow \textsc{Withdraw}(S, a)$;    if $succ$ then $\textsc{Deposit}(C, a)$;
```

Even if both DSs underlying C and S each individually satisfy DL, the whole banking DS does not: the amount a withdrawn from S can be lost if a crash occurs before it is deposited into C. What is needed for the correctness of this composition is the stronger *detectable recoverability* (or *detectability* in short) [Friedman et al. 2018]. Under detectability, after a crash a user can **(1)** detect if an operation was not invoked, interrupted by the crash, or completed before the crash; **(2)** resume the execution of an interrupted operation; and **(3)** retrieve the correct output for completed operations. If S and C were detectable, then one could detect the value withdrawn from S and whether it was deposited into C in case of a crash, and resume the interrupted operation.

Note that existing *persistent TM* (PTM) systems such as those of Krishnan et al. [2020]; Memaripour et al. [2017] provide such detectability guarantees. Specifically, code wrapped within a persistent transaction t is executed both atomically (i.e. it is thread-safe) and failure-atomically (i.e. either all or none of the effects of t take place in case of a crash, and thus t is detectable). Nevertheless, PTM systems have two limitations that make them unsuitable for implementing persistent concurrent DSs. First, the code wrapped within a PTM (or TM for that matter) is typically required to comprise simple memory read and write operations, rather than arbitrary operations associated with DSs. Second, even if one could enclose arbitrary DS operations within a PTM transaction, combining PTM and a concurrent DS is not straightforward: a PTM (as with TM) system provides its own concurrency control mechanism (e.g. via locking), clashing with and defeating the purpose of the already in place concurrency control mechanism of a concurrent DS. As such, PTM systems are not immediately suitable for implementing detectable concurrent DSs in PM.

Challenges. Our aim here is to devise a technique for implementing detectable concurrent DSs in PM in such a way that is both *generally applicable* (i.e. it can be applied to implement an arbitrary DS rather than tailored towards a specific DS, e.g. a queue) and *highly performant.*

This, however, is far from straightforward. Specifically, as we discuss below, although several detectable concurrent DSs have been proposed in the literature [Attiya et al. 2022, 2018; Ben-David et al. 2019; Friedman et al. 2018; Li and Golab 2021; Rusanovsky et al. 2021], to the best of our knowledge, each is either limited to simple algorithms or suffers from high runtime overhead.

- ***General Applicability***: Many of the existing detectable concurrent DSs are hand-tuned and manually reason about crash consistency and detectability. Friedman et al. [2018]; Li and Golab [2021] present detectable lock-free queues. Fatourou et al. [2022]; Rusanovsky et al. [2021] present a general combiner to construct persistent combining DSs, but it can recover only the last invocation of each operation. As such, in an execution of the banking example where S is withdrawn two times before a crash, we cannot distinguish whether the crash happened during the first or the second invocation of Withdraw. Attiya et al. [2018] present a detectable compare-and-swap (CAS) operation on PM locations as a general primitive operation for pointer-based DSs. However, the applicability of their CAS to concurrent DSs has not been established. Attiya et al. [2022] present a transformation from concurrent DSs in DRAM into those in PM with detectability, but this requires the operations to be strictly splittable into two phases: load-only *gather* and CAS-only *update*. Such a requirement is satisfied by data structures such as linked-lists [Harris 2001], but not by more sophisticated ones such as the Michael-Scott queue [Michael and Scott 1996] or hash tables [Chen et al. 2020; Shalev and Shavit 2006] that perform loads and CASes in an interleaved manner. Ben-David et al. [2019] present a more general transformation, but theirs requires the operations to follow specific patterns such as the *normalized* form [Timnat and Petrank 2014] for efficient transformation and makes a simplifying assumption that is not satisfied by real-world systems (see §7).
- ***High Performance***: While the overhead of detectability is modest or negligible for hand-tuned DSs [Attiya et al. 2022; Friedman et al. 2018; Li and Golab 2021; Rusanovsky et al. 2021], it is significant for the transformation of Ben-David et al. [2019] for two reasons. First, an object supporting a detectable CAS consumes $O(T)$ space in PM where T is the number of threads, prohibiting its use for space-efficient DSs such as hash tables and trees. More significantly, the detectable CAS object of Attiya et al. [2018] consumes $O(T^2)$ space in PM. Second, the transformed program writes and flushes to PM rather frequently (see §7 for details).

Contributions and Outline. To address the above challenges, we present MEMENTO: the first general programming framework for high-performance, detectable, concurrent DSs in PM.[1] To this end, we generalize Ramalingam and Vaswani [2013]'s type system that statically ensures the detectable recovery of programs in a simple core language. In contrast to the prior work, MEMENTO's type system additionally supports control constructs such as conditionals, loops, and function calls for general programming, and the CAS primitive operation for concurrent programming in PM. Our type system ensures programs to be deterministically replayed after a crash so that well-typed programs are detectably recoverable when simply re-executed *from the beginning* after a crash. As such, our type system substantially *reduces* the complexity of designing detectable DS in PM to that of designing volatile DS. Unlike most hand-tuned persistent DSs that require challenging-to-develop and reason-about DS-specific recovery code, our framework solely requires a program to conform to our type system, thereby *eliminating* the need for DS-specific recovery code! As example, we adapt several volatile concurrent DSs to well-typed programs and automatically derive detectable concurrent DSs. Specifically, we make the following contributions:

- In §2, we describe how to design programs that are deterministically replayed after a crash. We do so using two primitive operations, detectable checkpoint and CAS, by composing them with usual control constructs such as sequential composition, conditionals, and loops.
- In §3, we design a core language for persistent programming and its associated type system for deterministic replay, and prove that well-typed programs are detectably recoverable.
- In §4, we present an implementation of our core language in the Intel-x86 Optane DCPMM architecture. Our construction is not tightly coupled with Intel-x86 so that it can be adapted to other PM architectures like Samsung's Memory-Semantic SSD in a straightforward manner.
- In §5, we adapt several volatile, concurrent DSs to satisfy our type system, automatically deriving detectable concurrent DSs in PM. These include a lock-free linked-list [Harris 2001], Treiber stack [Treiber 1986], Michael-Scott queue [Michael and Scott 1996], a combining queue, and Clevel hash table [Chen et al. 2020]. In doing so, we capture the optimizations of hand-tuned persistent concurrent DSs with additional primitives and type derivation rules (§B and §C)[2], and support safe memory reclamation even in the presence of crashes (§D).
- In §6, we evaluate the detectability and performance of our CAS and automatically derived concurrent DSs in PM. They successfully recover from random thread and system crashes in stress tests, respectively (§6.1); and perform comparably with the existing hand-tuned persistent DSs with and without detectability (§6.2).

In §7, we conclude with related and future work. Our implementation and experimental results are open-sourced and available as supplementary material [Cho et al. 2023].

2 DESIGNING DETECTABLE PROGRAMS WITH DETERMINISTIC REPLAY

MEMENTO achieves detectability by deterministically replaying programs after a crash. Before presenting our type system that statically ensures deterministic replay of programs in §3, we first describe our key idea, which is recording the progress and result of a program using a *memento*, a thread-private log stored in PM (hence the framework name), in a compositional manner.

2.1 Ensuring Deterministic Replay of Composed Operations

Composition. Consider the TRANSFER function of our banking example (§1) shown in Algorithm 1: it attempts to withdraw *amount* from *savings* (L2), and if successful, it deposits the same amount into

[1]We use the word "concurrent" to emphasize MEMENTO's general applicability, but the framework applies not only to lock-free or lock-based concurrent CSs but also to sequential DSs.
[2]All alphabetical references in this paper refer to sections in the technical appendix [Cho et al. 2023].

Algorithm 1 Transfer from a savings account to a current account with mementos

```
1: function Transfer($savings, current, amount$, mid)                ▷ mid: memento id
2:     $succ \leftarrow$ Withdraw($savings, amount$, mid.withdraw);
3:     if $succ$ then Deposit($current, amount$, mid.desposit)
4: end function
```

current (L3). The code without highlighted parts is correct on volatile memory but not recoverable on PM in case of crashes. To ensure deterministic replay of Transfer, it suffices to ensure those of its sub-operations Withdraw and Deposit using sub-mementos mid.withdraw and mid.deposit, respectively. Regardless of whether the execution of a function f is finished or interrupted at crash time, thanks to its memento the post-crash re-execution of f will return the same result or resume from the interrupted program point, respectively. For instance, if the pre-crash execution crashes at L2, the post-crash re-execution resumes Withdraw thanks to its deterministic replay. On the other hand, if the pre-crash execution crashes during Deposit at L3, the post-crash re-execution produces the same result *succ* from Withdraw, takes the same branch, and resumes Deposit. In general, the deterministic replay property is preserved by sequential composition and conditionals.

Checkpoint Primitive. As a general-purpose primitive operation, our framework provides a detectable *checkpoint* operation that records the result of a read-only expression:

```
1: $v \leftarrow$ chkpt($\lambda.e$, mid)                ▷ $e$: read-only
```

Here, e is a read-only expression whose result may change across crashes due to, e.g. concurrent modifications to PM. The checkpoint operation first checks if a value is recorded in the memento mid, and if so it returns its value; otherwise, it executes e, records its result in the memento mid and returns the result. The checkpoint operation is detectable: even though it may partially execute e multiple times across crashes (hence the requirement for e to be read-only), it produces a unique result that is recorded in the memento across crashes and assigns this unique result to v.

A PM allocation is considered read-only as its effect is thread-local and becomes visible to other threads only after the address is published to shared memory. It is safe to leak PM allocations during crashes, as the underlying memory allocator is assumed to trace garbage after a crash.

Checkpoint operation is already proposed in prior work [Ben-David et al. 2019], but we generalize their implementation with *timestamps* (see §2.2 for details). We will present our design in §4.2.

Compare-and-Swap Primitive. As another general-purpose primitive operation for concurrent programming, our framework provides a detectable, persistent *compare-and-swap* (CAS) operation:[3]

```
1: $r \leftarrow$ pcas($loc, v_{old}, v_{new}$, mid)
```

This operation compares the current value of loc against v_{old}, and if the values match it updates it to v_{new}; otherwise the value of loc is unchanged. The return value $r \in \mathbb{B} \times \mathrm{Val}$ is a pair comprising a boolean flag reflecting whether the update was successful, and the original value held in loc. The operation guarantees that the result r is deterministic so long as the arguments are also deterministic. In particular, if a pcas were unsuccessful before a crash, its failure would be recorded in its memento mid and thus the post-crash execution would also fail by inspecting mid.

Note that deterministic replay cannot be achieved using plain CAS operations: in case of a crash, one loses such information as whether the plain CAS was performed, and if it was successful or not. The pcas requires additional synchronization in PM. Recognizing its general applicability, Attiya et al. [2018]; Ben-David et al. [2019] have proposed alternative implementations, but they consume

[3]Here we omit memory orderings [McKenney 2005]—e.g. release or acquire—but we annotate the most efficient and yet correct orderings in our implementation.

Algorithm 2 Insertion on the Harris concurrent sorted linked-list

```
1: function INSERT($head, val,$ mid)
2:     loop
3:         $(prev, next, blk) \leftarrow$ chkpt($\lambda.e_{\text{pnb}}$, mid.pnb);       ▷ timestamp: 30 | 80
4:         $succ \leftarrow$ pcas($prev$.next, $next, blk$, mid.cas);          ▷ timestamp: 20 | 90
5:         if $succ$ then return
6:     end loop
7: end function
   $e_{\text{pnb}} \triangleq (p, n) \leftarrow$ FIND($head, val$); $b \leftarrow$ palloc(⟨val : $val$; next : $n$⟩); return $(p, n, b)$
```

$O(T^2)$ and $O(T)$ PM space for each location, respectively, where T is the number of threads. By contrast, our implementation (§4.3) uses only 8 PM bytes for each location.

2.2 Supporting Simple Loops with Timestamps

The banking example uses a unique sub-memento for each sub-operation, making it easier to ensure a deterministic replay of composed operations. While feasible for simple programs, the unique memento assumption does not apply to complex programs with loops as the sub-mementos are reused across different loop iterations. To support loops, our framework employs *timestamps*.

Example: Concurrent Linked-List. Consider the INSERT operation on the concurrent sorted linked-list by Harris [2001] in Algorithm 2. For brevity, we omit the implementation of the function FIND($head, val$) (traversing the list from $head$ to find val) and the deallocation of non-inserted blocks (see §D for the implementation). As before, the code without highlighted parts is correct for volatile memory: it searches for adjacent blocks, $prev$ and $next$, between which val is inserted while preserving the sorted order and allocates a new block, blk, that contains val and points to $next$ (L3); performs a CAS on $prev.next$ from $next$ to blk (L4); and keeps trying until successful (L5).

Challenge: Reused Memento. Adding the highlighted parts (replacing cas with pcas at L4), programmers can ensure the deterministic replay of the loop body. However, it is insufficient to correctly recover from crashes after loop iterations as they reuse mementos. Consider an execution that crashes right after L3 in the *second* loop iteration. After the crash, mid.pnb contains the result of e_{pnb} in the *second* iteration, while mid.cas contains the result of the CAS in the *first* iteration. As such, it is necessary to distinguish the results of sub-operations from different iterations for correct recovery; otherwise, a post-crash execution would mix the sub-operation results.

To address the challenge of loops and more generally of complex control flow, the prior work performs additional writes and following flushes to PM to record the operation progress. Specifically, Attiya et al. [2018]; Li and Golab [2021] additionally reset memento-like "operation descriptors" by writing sentinel values to PM; and Ben-David et al. [2019] further checkpoint the program counter in PM. However, these additional writes and flushes to PM incur a significant performance overhead for high-contention workloads with heavy use of loops (see §6 for details).

Solution: Timestamp. To distinguish between the sub-operation results of different iterations efficiently (and record the operation progress more generally), our framework uses *timestamps*. A timestamp is a counter that increases monotonically during executions and across crashes.[4] Specifically, each primitive detectable sub-operation additionally records in its sub-memento the timestamp at which it completes. In the above scenario, the sub-operations may record timestamps

[4]Intel-x86 does not natively support such a timestamp with strong properties, but we develop such a counter in §4.2.

Algorithm 3 Resizing the Clevel hash table [Chen et al. 2020] (simplified)

```
1: function ResizeMoveArray(from, to, mid)
2:     loop
3:         $i \leftarrow$ chkpt($\lambda.\phi(0, i+1)$, mid.i);                ▷ timestamp: 30 | 80
4:         if $i \geq |from|$ then return
5:         ResizeMoveEntry(from, i, to, mid.entry)               ▷ timestamp: 20 | 90
6:     end loop
7: end function
```

of 10 and 20 in mid.pnb and mid.cas in the first loop iteration, respectively, and then overwrite the timestamp of 30 in mid.pnb in the next iteration.

In the post-crash execution, our framework first observes that timestamp 30 in mid.pnb and then 20 in mid.cas, which is *not* monotonically increasing with the control flow. That is, the checkpoint at L3 was performed in the last iteration before the crash, but the pcas at L4 was not. As such, the post-crash execution may resume at L4 and re-execute pcas.

Regardless of the program point at which the execution crashes, the post-crash execution can deterministically replay the last iteration before the crash. Suppose the timestamps recorded in mid.pnb and mid.cas were 80 and 90, respectively. Then the post-crash execution replays the last iteration by observing the monotonically increasing timestamps (80 at L3 and 90 at L4) and retrieves the recorded results. Thereafter, it will either successfully return or try again (L5).

Unlike prior approaches [Attiya et al. 2018; Ben-David et al. 2019], our approach does not incur additional writes and flushes to PM.[5] On the one hand, our primitive operations, checkpoint and CAS, record an operation's timestamp and result atomically at once. On the other hand, our framework does not require additional writes and flushes for loops and other control constructs.

2.3 Supporting Loop-Carried Dependence by Checkpointing Dependent Variables

In the presence of loop-carried dependence, timestamps alone do not guarantee deterministic replay because dependent variable values may be lost in case of a crash. As such, our framework further requires programmers to checkpoint the dependent variables for each iteration.

Example: Clevel Hash Table. Consider the ResizeMoveArray operation on the Clevel hash table [Chen et al. 2020] presented in Algorithm 3. When resizing the hash table, every entry in the array of an old level, *from*, is moved to the array of a new level, *to*. To do this, the operation iterates over *from* (L3) and invokes the sub-operation ResizeMoveEntry for each entry index i (for brevity we omit ResizeMoveEntry). To reveal loop-carried dependence explicitly, we represent the code in the Static Single Assignment (SSA) form [Cytron et al. 1989, 1991].[6] In the SSA form, loop-dependent variables are defined as a ϕ-*node* at the beginning of the loop. A ϕ-node of the form $v = \phi(v_0, v_1)$ assign v_0 (resp. v_1) to v if it is the first (resp. a later) iteration. In our example of Algorithm 3, i gets 0 in the first iteration and $i + 1$ in the later iterations at L3.

Challenge: Dependent Variable. With the highlighted part, especially invoking the sub-operation with an additional memento argument mid.entry at L5, our framework ensures the deterministic replay of the loop body. However, the loop-dependent variable i makes it challenging to correctly recover from crashes because the framework needs to restore the value of i in the last iteration.

[5]Since our approach reduces the number of writes as well as that of flushes, it has performance advantages over the prior approaches in a wide range of PM platforms including Intel eADR [Intel 2021].
[6]The SSA form can represent a much more general class of control flow-dependent variables than loop-dependent variables [Cytron et al. 1989, 1991]. Although we present this example in SSA form, we do not require SSA in our implementation.

$$\begin{array}{rll}
p & ::= [\delta]\ \vec{s_1}\ ||\ \ldots\ ||\ \vec{s_n} & \textit{program} \\
s \in \mathrm{Stmt} & ::= r \leftarrow e \mid r \leftarrow \mathtt{pload}(e) \mid r \leftarrow \mathtt{palloc}(e) & \textit{assignment \& PM} \\
& \mid \mathtt{if}\ (e)\ \vec{s_\mathrm{t}}\ \vec{s_\mathrm{f}} \mid \mathtt{loop}\ r\ e\ \vec{s} \mid \mathtt{continue}\ e \mid \mathtt{break} & \textit{control constructs} \\
& \mid r \leftarrow f(\vec{e}) \mid \mathtt{return}\ e & \textit{function call/return} \\
& \mid r \leftarrow \mathtt{chkpt}(\vec{s}, e_\mathrm{mid}) \mid r \leftarrow \mathtt{pcas}(e_\mathrm{loc}, e_\mathrm{old}, e_\mathrm{new}, e_\mathrm{mid}) & \textit{detectable op.} \\
e \in \mathrm{Expr} & ::= () \mid z \mid b \mid \mathit{mid} \mid r \mid (e_1\ \mathit{op}\ e_2) \mid e.i \mid (e_1, e_2) \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e & \textit{pure expr.} \\
& \mid \mathsf{match}\ e\ \{\ \mathsf{inl}\ e_\mathrm{l} \Rightarrow e'_\mathrm{l}\ ,\ \mathsf{inr}\ e_\mathrm{r} \Rightarrow e'_\mathrm{r}\} \mid \epsilon \mid e.\mathit{lab} \mid\ \ldots & \\
v \in \mathrm{Val} & ::= () \mid z \mid b \mid \mathit{mid} \mid (v_1, v_2) \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v & \textit{value}
\end{array}$$

$$z \in \mathbb{Z} \qquad b \in \mathbb{B} \qquad op \in \mathrm{Op} \qquad r \in \mathrm{VReg} \triangleq \mathbb{N} \qquad \sigma \in \mathrm{VRegMap} \triangleq \mathrm{VReg} \rightharpoonup \mathrm{Val}$$

$$f \in \mathrm{FnId} \qquad \delta \in \mathrm{Env} \triangleq \mathrm{FnId} \rightharpoonup (\overrightarrow{\mathrm{VReg} \times \mathrm{Stmt}}) \qquad l \in \mathrm{PLoc} \triangleq \mathbb{N} \qquad \mathit{lab} \in \mathrm{Label} \qquad \mathit{mid} \in \overrightarrow{\mathrm{Label}}$$

$$t \in \mathrm{Time} \triangleq \mathbb{N} \qquad \mathit{tid} \in \mathrm{TId} \triangleq \mathbb{N} \qquad \mathit{mmts} \in \mathrm{Mmts} \triangleq \overrightarrow{\mathrm{Label}} \longrightarrow \langle \mathrm{val}:\mathrm{Val}; \mathrm{time}:\mathrm{Time}\rangle$$

$$c \in \mathrm{Cont} \qquad ts \in \mathrm{TState} \triangleq \langle \mathrm{regs} : \mathrm{VRegMap}; \mathrm{time}:\mathrm{Time}\rangle \qquad T \in \mathrm{Thread} \triangleq \overrightarrow{\mathrm{Stmt}} \times \overrightarrow{\mathrm{Cont}} \times \mathrm{TState} \times \mathrm{Mmts}$$

$$ev \in \mathrm{Event}\ ::=\ \mathsf{R}(l, v)\ \mid\ \mathsf{U}(l, v_{old}, v_{new}) \qquad tr \in \overrightarrow{\mathrm{Event}} \qquad \mathsf{mem} \in \mathrm{Mem} \qquad M \in \mathrm{Machine} \triangleq \overrightarrow{\mathrm{Thread}} \times \mathrm{Mem}$$

$$\boxed{\vec{s_1}, \vec{c_1}, ts_1, mmts_1 \xrightarrow{tr}_{\delta} \vec{s_2}, \vec{c_2}, ts_2, mmts_2} \qquad \boxed{\mathsf{mem}_1 \xrightarrow{tr} \mathsf{mem}_2} \qquad \boxed{M_1 \xrightarrow{tr}_{p} M_2}$$

(MACHINE-STEP)

$$\frac{\mathcal{T}_1[tid] = (\vec{s_1}, \vec{c_1}, ts_1, mmts_1) \quad \mathcal{T}_2 = \mathcal{T}_1[tid \mapsto (\vec{s_2}, \vec{c_2}, ts_2, mmts_2)] \\ \vec{s_1}, \vec{c_1}, ts_1, mmts_1 \xrightarrow{tr}_{p.\delta} \vec{s_2}, \vec{c_2}, ts_2, mmts_2 \quad \mathsf{mem}_1 \xrightarrow{tr} \mathsf{mem}_2}{(\mathcal{T}_1, \mathsf{mem}_1) \xrightarrow{tr|_{\mathsf{U}}}_{p} (\mathcal{T}_2, \mathsf{mem}_2)}$$

(MACHINE-CRASH)

$$\frac{\mathcal{T}_1[tid] = (\vec{s_1}, \vec{c_1}, ts_1, mmts_1) \\ \mathcal{T}_2 = \mathcal{T}_1[tid \mapsto (\overrightarrow{p.s_{tid}}, [], ts_{\mathrm{init}}, mmts_1)]}{(\mathcal{T}_1, \mathsf{mem}) \xrightarrow{[]}_{p} (\mathcal{T}_2, \mathsf{mem})}$$

Fig. 1. The syntax and semantics of our core PM language (excerpt)

Solution: Checkpoint at the Loop Head. To address the challenge above, our framework requires programmers to checkpoint dependent variables, e.g. i, at the loop head. In the post-crash execution, the checkpoint operation retrieves the i value in the last iteration and, moreover, *delimits* the last iteration. For instance, suppose that L3 and L5 record timestamps 30 and 20, respectively. Then the last iteration began at timestamp 30 and the post-crash execution should re-execute L5. Similarly, if L3 and L5 respectively record timestamps 80 and 90, then the last iteration began at timestamp 80 and the post-crash execution should retrieve the sub-operation result recorded in mid.entry at L5.

In the presence of multiple dependent variables, our framework requires programmers to merge them all into a single tuple or struct and checkpoint it at once. Otherwise, dependent variables of two consecutive iterations can be mixed. For instance, suppose there were two dependent variables, x and y, and they were individually checkpointed. If only x were checkpointed at the loop head and then the thread crashes, then the post-crash execution retrieves the value of x from the last iteration and that of y from the previous iteration, violating the recovery correctness.

3 TYPE SYSTEM FOR DETECTABILITY

We next formalize the key idea presented in §2. We design a core language for PM (§3.1) and a type system for deterministic replay (§3.2), and prove that typed programs are detectable (§3.3).

3.1 Core Language

We present the syntax and semantics of our core language for PM in Fig. 1. We discuss the implementation of our language later in §4, and give its semantics in the technical appendix (§F).

A program, p, consists of a function environment, δ, and a list of statements, $\overrightarrow{s_{tid}}$, for each thread tid. An assignment statement, $r \leftarrow e$ where $r \in$ VReg is a register id and $e \in$ Expr is a pure expression, evaluates e to a value in Val $\subseteq$ Expr and assigns it to r. An expression is either a constant, register, arithmetic/boolean operation, tuple/union introduction/elimination, *memento id*, empty expression (ϵ) or concatenation ($e.lab$, see below). A value is an irreducible expression without variables. A load statement, $r \leftarrow$ `pload`(e), evaluates e as a PM location, $l \in \text{PLoc} \triangleq \mathbb{N}$, in the *shared* memory, loads the value of l and writes it to r. For simplicity, we classify PM locations into shared and thread-local ones so that we can use the former as concurrent DS memory blocks and the latter as mementos. An allocation, $r \leftarrow$ `palloc`(e), initializes a fresh PM location in the shared memory with the value evaluated from e and writes the location to r.

A conditional statement, `if` (e) $\vec{s_t}$ $\vec{s_f}$, reduces either to $\vec{s_t}$ or to $\vec{s_f}$ depending on the value evaluated from e. Loops reveal loop-carried dependence explicitly in the style of the SSA form (§2.3). Specifically, `loop` r e $\vec{s}$ **(1)** evaluates the initial value from e and assigns it to the dependent variable r; **(2)** executes the body $\vec{s}$; **(3)** in doing so, if `continue` e is executed, then the (merged) loop-carried dependent value evaluated from e is assigned to r, and $\vec{s}$ is re-executed for the next iteration; and **(4)** if `break` is executed, the loop terminates. A function call, $r \leftarrow f(\vec{e})$, evaluates the arguments $\vec{e}$, finds the function id f in the program's function environment δ with $\delta(f) = (\overrightarrow{prms}, \vec{s_f}) \in \overrightarrow{\text{VReg}} \times \overrightarrow{\text{Stmt}}$, and executes the function body $\vec{s_f}$ with a fresh variable context assigning the evaluated arguments to $\overrightarrow{prms}$. If `return` e is executed, then the control goes back to the caller and the return value evaluated from e is assigned to r.

We treat primitive detectable operations as language constructs and implement them on Intel-x86 later in §4. Primitive detectable operations comprise `chkpt` and `pcas`. A detectable checkpoint, $r \leftarrow$ `chkpt`$(\vec{s}, e_{\mathsf{mid}})$, evaluates $\vec{s}$ as if it is a function body, but using the same variable context as the operation's caller as a variable-capturing closure. A detectable CAS, $r \leftarrow$ `pcas`$(e_{\mathsf{l}}, e_{\mathsf{o}}, e_{\mathsf{n}}, e_{\mathsf{mid}})$, evaluates the expressions respectively to v_{l}, v_{o}, and v_{n}, attempts to update the PM location v_{l} from v_{o} to v_{n} atomically, and writes whether it succeeded to r. For both `chkpt` and `pcas`, their results and timestamps are checkpointed at the thread's sub-memento (located in its private PM) identified by the memento id (mid) evaluated from e_{mid}.

A thread consists of statements ($\vec{s}$), loop and function continuations ($\vec{c}$, definition omitted), a volatile state (ts), and a persistent memento ($mmts$). Continuations are pushed (resp. popped) for loop and call (resp. break and return) statements, respectively. A thread state, ts, consists of a register file (ts.regs) and the thread's last observed timestamp (ts.time, see §2.2). To maintain its invariant, ts.time is initialized with zero at thread initialization point (see machine-crash below), and incremented when a primitive operation is executed or replayed. When executing a primitive operation op, we compare ts.time with the timestamp t_{mmt} checkpointed in the memento of op. If $ts.\text{time} < t_{\mathsf{mmt}}$, then op was executed before the crash, and thus we simply update ts.time to t_{mmt}; otherwise, the replay is over and we execute op and update ts.time to a new timestamp. A memento is a map from memento ids (lists of labels) to primitive mementos that record values and timestamps; e.g. the id list.pnb denotes the primitive memento used at L3 in Algorithm 2. In our implementation, we statically reason about the structure and size of the memento for each operation with types. Lastly, a machine, M, consists of a list of threads ($\mathcal{T}$) and a memory (mem).

A judgement of the form $\vec{s_1}, \vec{c_1}, ts_1, mmts_1 \xrightarrow{tr}_{\delta} \vec{s_2}, \vec{c_2}, ts_2, mmts_2$ denotes a *thread transition* for environment δ, emitting a *trace* tr. A trace is a list of *events*; an event is either a read ($\mathsf{R}(l, v)$, reading v from shared PM location l) or an update ($\mathsf{U}(l, v_{old}, v_{new})$, atomically updating l from v_{old} to v_{new}). For read events, the values read from the shared memory are constrained not by thread transitions but by *memory transitions* of the form $\text{mem}_1 \xrightarrow{tr} \text{mem}_2$. Two transitions are combined into a *machine transition* of the form $M_1 \xrightarrow{tr}_p M_2$ for program p. The machine-step rule states that

$$labs \in \mathcal{P}(\text{Label}) \qquad \text{FnType} ::= \mathsf{RO} \mid \mathsf{RW}$$
$$\Delta \in \text{EnvType} \triangleq \text{FnId} \rightarrow \text{FnType}$$

$$\boxed{\vdash p} \qquad \textbf{(program)}\ \frac{\vdash \delta : \Delta \qquad \Delta \vdash_{labs_{tid}} \vec{s_{tid}} \quad \text{for each } tid}{\vdash [\delta]\ \vec{s_1} \mid\mid \ldots \mid\mid \vec{s_n}}$$

$$\boxed{\vdash \delta : \Delta} \qquad \textbf{(env-empty)}\ \frac{}{\vdash\ : []} \qquad \textbf{(env-ro)}\ \frac{\vdash \delta : \Delta \qquad \Delta \vdash_{\mathsf{RO}} \vec{s}}{\vdash \delta[f \mapsto (\vec{prms}, \vec{s})] : \Delta[f \mapsto \mathsf{RO}]}$$

$$\textbf{(env-rw)}\ \frac{\vdash \delta : \Delta \qquad \Delta \vdash_{labs} \vec{s} \qquad \vec{prms_{\mathsf{all}}} = \vec{prms} +\!\!+ \{\mathsf{mid}\}}{\vdash \delta[f \mapsto (\vec{prms_{\mathsf{all}}}, \vec{s})] : \Delta[f \mapsto \mathsf{RW}]}$$

$$\boxed{\Delta \vdash_{labs} \vec{s}} \qquad \textbf{(empty)}\ \frac{}{\Delta \vdash_{\emptyset} []} \qquad \textbf{(assign)}\ \frac{}{\Delta \vdash_{\emptyset} [r \leftarrow e]} \qquad \textbf{(cas)}\ \frac{}{\Delta \vdash_{\{lab\}} [r \leftarrow \mathsf{pcas}(e_{\mathsf{l}}, e_{\mathsf{o}}, e_{\mathsf{n}}, \mathsf{mid}.lab)]}$$

$$\textbf{(chkpt)}\ \frac{\Delta \vdash_{\mathsf{RO}} \vec{s}}{\Delta \vdash_{\{lab\}} [r \leftarrow \mathsf{chkpt}(\vec{s}, \mathsf{mid}.lab)]} \qquad \textbf{(seq)}\ \frac{labs_{\mathsf{l}} \cap labs_{\mathsf{r}} = \emptyset \quad \Delta \vdash_{labs_{\mathsf{l}}} \vec{s_{\mathsf{l}}} \quad \Delta \vdash_{labs_{\mathsf{r}}} \vec{s_{\mathsf{r}}}}{\Delta \vdash_{labs_{\mathsf{l}} \uplus labs_{\mathsf{r}}} \vec{s_{\mathsf{l}}} +\!\!+ \vec{s_{\mathsf{r}}}}$$

$$\textbf{(if-then-else)}\ \frac{\Delta \vdash_{labs_{\mathsf{t}}} \vec{s_{\mathsf{t}}} \qquad \Delta \vdash_{labs_{\mathsf{f}}} \vec{s_{\mathsf{f}}}}{\Delta \vdash_{labs_{\mathsf{t}} \cup labs_{\mathsf{f}}} [\mathtt{if}\ (e)\ \vec{s_{\mathsf{t}}}\ \vec{s_{\mathsf{f}}}]}$$

$$\textbf{(loop-simple)}\ \frac{\Delta \vdash_{labs} \vec{s}}{\Delta \vdash_{labs} [\mathtt{loop}\ \_\ ()\ \vec{s}]} \qquad \textbf{(loop)}\ \frac{\Delta \vdash_{labs} \vec{s} \qquad lab \notin labs}{\Delta \vdash_{\{lab\} \uplus labs} [\mathtt{loop}\ r\ e\ ((r \leftarrow \mathsf{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \vec{s})]}$$

$$\textbf{(continue)}\ \frac{}{\Delta \vdash_{\emptyset} [\mathtt{continue}\ e]} \qquad \textbf{(break)}\ \frac{}{\Delta \vdash_{\emptyset} [\mathtt{break}]} \qquad \textbf{(call)}\ \frac{\Delta(f) = \mathsf{RW}}{\Delta \vdash_{\{lab\}} [r \leftarrow f(\vec{e} +\!\!+ \{\mathsf{mid}.lab\})]} \qquad \textbf{(return)}\ \frac{}{\Delta \vdash_{\emptyset} [\mathtt{return}\ e]}$$

Fig. 2. Type System (excerpt)

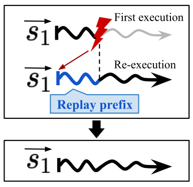
a thread may execute a step tr, transitioning the memory with the same trace tr, emitting only updates externally ($tr|_{\mathsf{U}}$); the MACHINE-CRASH states that a thread may crash and re-execute the initial statements with an empty continuation, initial thread state, and the preserved memento.

3.2 Type System

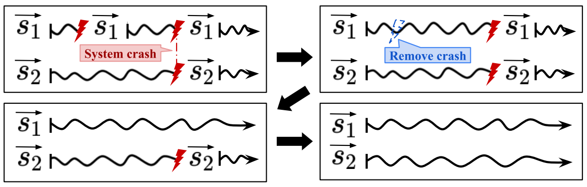
We present our type system for detectable operations with deterministic replay in Fig. 2. The PROGRAM rule states that a program is typed if its function environment and each thread's statements are typed. A judgement of the form $\vdash \delta : \Delta$ denotes that for each function id f, the function $\delta(f)$ is detectable with type $\Delta(f) \in$ FnType. A function type is either RO, meaning the function only reads from shared PM locations and does not access mementos at all; or RW, meaning the function reads and writes to shared PM locations and accesses only those mementos prefixed by mid given as its last argument. The ENV-EMPTY rule states that the empty function environment is typed; ENV-RO adds a read-only function to the environment[7]; and ENV-RW adds a read-write function with the last parameter being the memento id mid. The judgement $\Delta \vdash_{labs} \vec{s}$ in the premise of ENV-RW states that for any function environment (δ) with type Δ, the execution of $\vec{s}$ satisfies the interpretation of RW while using only those sub-mementos prefixed by mid.lab for some $lab \in labs$.

For read-write functions, EMPTY states that the empty statement list is typed for any function environment type (Δ) using no mementos ($\emptyset$); and ASSIGN, CONTINUE, BREAK and RETURN state that so are assignment, continue, break, and return statements for all sub-expressions as they are

[7]For brevity, we omit the definition of the read-only $\vdash_{\mathsf{RO}}$ judgement as it is straightforward (see §G for the its definition).



(a) Deterministic Replay



(b) Removing Crashes with Deterministic Replay

Fig. 3. Proving detectability by gradually removing crashes

pure.[8] The SEQ composes lists of statements so long as they use disjoint mementos ($labs_l \cap labs_r = \emptyset$) and sequential composition uses their disjoint union ($labs_l \uplus labs_r$). The IF-THEN-ELSE composes a conditional branch without requiring disjointness as only one branch is executed (see §2.1).

The CAS rule states that a pcas is typed against the memento label it uses (lab); the CHKPT behaves analogously so long as the checkpoint body ($\vec{s}$) is read-only. We require the body's result to be immediately checkpointed before being assigned to a register for deterministic replay. For instance, consider an execution of Algorithm 2 where among $prev$, $next$ and blk obtained at L3only $prev$ is checkpointed before a crash. The post-crash execution then re-calculates new values, $(prev', next', blk')$, and uses the old $prev$ from the memento but the new values $next'$, blk', mixing the results of different executions across crashes. This leads to a bug: as list traversal is non-deterministic, $prev$ and $next'$ may not be adjacent to each other, breaking the list invariant.

The LOOP-SIMPLE states that a loop without loop-carried dependence is typed if its body is ($\vec{s}$). Here, the loop-dependent variable "_" means it is written to nowhere, or equivalently, there are no dependent variables (§2.2). The LOOP states that a loop is typed if so is its body, its dependent variable (r) is checkpointed at the loop head, and the checkpoint and body use disjoint memento labels (§2.3). The CALL states that an RW function call is typed against the memento label it uses.

3.3 Detectability of Typed Programs

We sketch the proof of the detectability of typed programs and give the full proof in §H. Unlike the prior work [Attiya et al. 2018; Friedman et al. 2018], we formulate detectability in terms of behaviour refinement. For a program, p, we say event trace tr is a *behaviour* of p, written $tr \in B^{\lightning}(p)$, if there exists M such that $init(p) \xrightarrow{tr}{}^{*}_{p} M$, where $init(p)$ is the initial machine of p and $\xrightarrow{tr}{}^{*}_{p}$ is the reflexive transitive closure of the machine transition $\xrightarrow{tr}_{p}$ with concatenated event traces. A tr is a *crash-free behaviour* of p, written $tr \in B(p)$ if it is a behaviour from a crash-free machine execution using only MACHINE-STEP. We then prove the following theorem.

THEOREM 3.1 (DETECTABILITY). *Given a program p, if $\vdash p$ holds, then $B^{\lightning}(p) \subseteq B(p)$.*

This theorem ensures failure transparency in that crashes do not introduce additional behaviours; that is, this theorem ensures the detectable recoverability of typed programs.

We prove this theorem by gradually transforming an arbitrary execution of p into one without crashes while preserving the behaviour, as illustrated in Fig. 3. We exploit the fact that each thread interacts with the other components only via event traces: as long as event traces are preserved, we can *locally merge* a thread's consecutive executions across crashes into one without crashes.

[8]We do not establish the usual soundness result with our type system; e.g. while we can derive $\Delta \vdash_{\emptyset} [r_1 \leftarrow r_2]$ for any Δ, r_1 and r_2, the $r_1 \leftarrow r_2$ may get stuck as r_2 is a free variable. Though it is straightforward to adapt our system for soundness, we forgo this as this is not our aim and our type system is sufficient for our main goal: detectability by deterministic replay.

Subsequently, the resulting machine execution would produce the same behaviour as before with fewer crashes. Going forward, we will get a crash-free execution with the same behaviour.

Deterministic Replay. We formulate the ability to locally merge thread executions in Def. 3.2. We assume that a thread executes the statements $\vec{s}$ twice, before and after a crash. As such, the statements, continuations, and volatile thread state are initialized and the memento ($mmts_\omega$) is preserved. There then is an execution without crashes that results in the same memento ($\underline{mmts_\omega}$) while emitting an event trace (tr_x) that *refines* the original event trace ($tr \mathbin{+\!\!+} \underline{tr}$): we can reach tr_x from $tr \mathbin{+\!\!+} \underline{tr}$ by removing some read events. Trace refinement is sufficient to replace thread executions in a machine execution while preserving its behaviour, because machine transitions ignore read events and memory transitions are closed under trace refinement.

Definition 3.2 (Deterministic Replay). Let δ be a function environment and $\vec{s}$ be a list of statements. We say $\vec{s}$ is deterministically replayed for δ, denoted by $\mathsf{DR}(\delta, \vec{s})$, if the following holds:

$$\forall tr, \underline{tr}, \vec{s_\omega}, \underline{\vec{s_\omega}}, \vec{c_\omega}, \underline{\vec{c_\omega}}, ts, ts_\omega, \underline{ts_\omega}, mmts, mmts_\omega, \underline{mmts_\omega}.$$

$$\vec{s}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \vec{s_\omega}, \vec{c_\omega}, ts_\omega, mmts_\omega \longrightarrow \vec{s}, [], ts, mmts_\omega \xrightarrow{\underline{tr}}{}^{*}_{\delta} \underline{\vec{s_\omega}}, \underline{\vec{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \longrightarrow$$

$$\exists tr_x, \vec{s_x}, \vec{c_x}, ts_x.\ \vec{s}, [], ts, mmts \xrightarrow{tr_x}{}^{*}_{\delta} \vec{s_x}, \vec{c_x}, ts_x, \underline{mmts_\omega} \ \wedge\ tr_x \sim tr \mathbin{+\!\!+} \underline{tr}\ .$$

Lemma 3.3. *Let δ be an environment, Δ be an environment type, $\vec{s}$ be a list of statements, and labs be a set of labels. If we have $\vdash \delta : \Delta$ and $\Delta \vdash_{labs} \vec{s}$, then* $\mathsf{DR}(\delta, \vec{s})$.

This lemma states that typed statements are deterministically replayed. We prove it by strong induction on the derivations of $\vdash \delta : \Delta$ and $\Delta \vdash_{labs} \vec{s}$, formalizing the arguments presented in §2.

Erasure. In the absence of crashes, a program p behaves equivalently to the *erasure* of p, written $erase(p)$, intuitively corresponding to removing the highlighted parts in §2. In particular, memento parameters and arguments are removed, checkpoint operations are removed, and `pcas` operations are replaced with plain `cas` operations. We thus obtain the following theorem.

Theorem 3.4 (Erasure). *Given a program p, If $\vdash p$ holds, then $B^{\lightning}(p) \subseteq B(erase(p))$.*

The theorem effectively reduces the complexity of designing detectable and persistent DS to that of designing volatile DS (already well-studied) and adapting volatile DS to our type system (straightforward). In particular, programmers no longer need to write challenging-to-develop and reason-about DS-specific recovery code, which is required by most hand-tuned persistent DSs. This way, we will straightforwardly design a wide variety of high-performance detectable DSs in §5.

4 IMPLEMENTATION OF THE CORE LANGUAGE

To show the feasibility and practicality of our core language in §3, we implement it on Intel-x86.

4.1 Framework

PM Primitive. We use the App Direct mode of Intel-x86 Optane DCPMM to access PM locations with byte addressability via load, store, and CAS instructions. We use **clwb** instructions to ensure a write to a PM location is persisted: a store or CAS to a PM cache line *cl* is guaranteed to be persisted if followed by **clwb** *cl* and then **sfence**, **mfence**, or a successful CAS. We refer the reader to Cho et al. [2021]; Raad et al. [2019] for the formal semantics of **clwb**. We use Ralloc [Cai et al. 2020] for PM allocation and our modified version of Crossbeam [Crossbeam 2022] for safe memory reclamation of shared PM locations (see §D for more details on reclamation).

Crash Handler. To emulate MACHINE-CRASH, we install a *crash handler* that continuously observes and handles crashes. **(1)** When a thread crashes, which may happen due to signals but not widely considered in the literature [Attiya et al. 2022, 2018; Ben-David et al. 2019], the handler creates a new thread that executes the original thread's initial statements. It also initializes the thread state (ts), e.g. setting ts.time to zero, and runtime resources such as reclamation handle (see §D for more details). **(2)** When the whole system crashes, the post-crash execution first executes the handler, which then initializes the system state *as if* every thread experiences just a thread crash instead of the system crash. Specifically, the handler performs Ralloc's garbage collection, initializes volatile data used by primitive operations (see §4.2 and §4.3 for details), and revives the threads.

Timestamp. The core language assumes a consistent clock for multiple threads across crashes. We design such a clock on Intel-x86 using the `rdtscp` instruction generating hardware timestamps. The hardware clock is consistent for a single thread: *strictly increasing* and *serializing* in that `rdtscp` followed by **lfence** is not reordered with the surrounding instructions [Intel 2022].

However, Kashyap et al. [2018] observe that the clock is *not* consistent for multiple threads across crashes as follows. **(1)** The clock is reset to zero when the machine is rebooted after a crash. **(2)** The clock has an inter-core skew due to misaligned delivery of the `RESET` signal at the system boot. As such, even if an `rdtscp` instruction *happens before* [Owens et al. 2009] another in a different thread, their timestamps may not be ordered. Still, the skew is *invariant*: constant regardless of dynamic frequency and voltage scaling. For the core language, we address these caveats as follows.

For reset, the crash handler calibrates the clock at the system boot. Specifically, it ❶ calculates the maximum timestamp checkpointed in all mementos, t_{max}; ❷ generates the current timestamp, t_{init}, using `rdtscp`; and ❸ adds offset $(-t_{\text{init}} + t_{\text{max}})$ to all timestamps generated by `rdtscp`. The calibrated timestamps are then always larger than those checkpointed before the system boot.

For skew, we relax the synchronization criteria of the clock. We follow Kashyap et al. [2018] to measure the maximum pair-wise inter-core skew, O_g. We then make the following observation.

OBSERVATION 1 (WEAK GLOBAL SYNCHRONIZATION). *Suppose a and b are* `rdtscp; lfence` *instruction sequences. If either* $a \xrightarrow{po} b$ *(single-thread program order) or* $a \xrightarrow{hb} \mathbf{wait}(O_g) \xrightarrow{hb} b$ *(multi-thread happens-before), then a's timestamp is less than b's.*

Here, $\mathbf{wait}(O_g)$ is a spin loop to provide a sufficient margin for the clock skew. The single-thread program order and multi-thread happens-before order conditions are used in the implementation of checkpoint (§4.2) and CAS (§4.3) operations, respectively.

The single-thread program order sufficiently separates two `rdtscp` instructions even if the thread is context-switched in-between. Even if the thread switches to a core with a negative timestamp offset, its effect, bounded by O_g (60ns at the maximum in our evaluation), is subsumed by the context switch latency (2-5µs at the minimum [Blandy 2022; Microsoft 2023]). Similarly, for the multi-thread happens-before condtition, O_g sufficiently separates two `rdtscp` instructions regardless of their executed cores because O_g is the maximum *inter-core* skew.

4.2 Detectable Checkpoint

We implement the `chkpt` operation of the core language (§3.1) on Intel-x86. Following Ben-David et al. [2019], we ensure the atomicity of `chkpt` (i.e. one never observes a partially checkpointed value) by double buffering: while a buffer is being written, the other buffer holds a valid value. Moreover, we record timestamps and values in PM to deterministically replay control flow (§2.2).

We present our implementation in Algorithm 4. To atomically update a timestamped value in the *abstract* memento (§3.1), its *concrete* implementation uses two timestamped values, *stale* and *latest*. The algorithm then ❶ compares the given memento's two timestamps (st and lt) to distinguish

Algorithm 4 Detectable checkpoint

```
1:  function chkpt($\vec{s}$, mid)
2:      $t_0 \leftarrow \text{LOAD}_{\text{PLN}}(mmts[\text{mid}][0].\text{time})$
3:      $t_1 \leftarrow \text{LOAD}_{\text{PLN}}(mmts[\text{mid}][1].\text{time})$
4:      $t_{\text{mmt}} \leftarrow (t_0 < t_1)\ ?\ t_1 : t_0$
5:      $(st, lt) \leftarrow (t_0 < t_1)\ ?\ (0,1) : (1,0)$

6:      if $t_{\text{mmt}} > ts.\text{time}$ then
7:          $ts.\text{time} \leftarrow t_{\text{mmt}}$
8:          $v_r \leftarrow \text{LOAD}_{\text{PLN}}(mmts[\text{mid}][lt].\text{val})$
9:          return $v_r$
10:     end if
11:     $v_r \leftarrow$ exec $\vec{s}$
12:     $\text{STORE}_{\text{PLN}}(mmts[\text{mid}][st].\text{val}, v_r)$
13:     if $\text{SIZEOF}(mmt) > CLSIZE$ then
14:         flushopt $mmts[\text{mid}][st].\text{val}$; sfence
15:     end if
16:     $t \leftarrow$ rdtscp
17:     $\text{STORE}_{\text{PLN}}(mmts[\text{mid}][st].\text{time}, t)$
18:     flushopt $mmts[\text{mid}][st].\text{time}$; sfence
19:     $ts.\text{time} \leftarrow t$
20:     return $v_r$
21: end function
```

stale from latest (L2-L5); ❷ if the memento's timestamp (t_{mmt}) is greater than the thread's replaying timestamp (ts.time), then the operation was already performed before the crash. In this case, ts.time is incremented to t_{mmt} first, and then the pre-crash result is replayed by simply returning the old returned value (L6-10); ❸ write the result of the given statements to the memento's stale buffer (L12); ❹ flush the stale buffer, unless the memento fits in a cache line so that the buffer is anyway flushed at L18, following van Renen et al. [2020]'s optimization technique (L14); and ❺ update the stale timestamp to the current timestamp (L17), flush it (L18), update ts.time (L19), and return the result (L20). Here, "**flushopt** l" is a shorthand for performing **clwb** cl on all cache lines cl that spanning location l.

4.3 Detectable Compare-and-Swap

We implement the `pcas` operation of our core language (§3.1) on Intel-x86. Following Attiya et al. [2018]; Ben-David et al. [2019], our `pcas` on location l comprises three phases: *locking* l with an architecture-provided plain CAS, *committing* the operation with PM writes, and *unlocking* l with another plain CAS. If a thread observes a locked location, it *helps* the ongoing operation to guarantee lock freedom. When helping, it is crucial to notify such a fact to the helped thread to ensure deterministic replay; otherwise, in the case that a thread performs a locking CAS, crashes, and gets helped, then the thread would incorrectly perform the same CAS (that is already performed by the helper) again in the post-crash execution. While the helping mechanism of Attiya et al. [2018] (resp. Ben-David et al. [2019]) requires an array of $O(T^2)$ (resp. $O(T)$) sequence numbers in PM for each location (where T is the number of threads), we reduce the space consumption in PM to only 8 bytes per location. The key idea is comparing timestamps as for loops (§2.2).

Components. An 8-byte location consists of 1-bit *parity* for helping, 1-bit *helping flag* to prevent ABA[9], 8-bit thread id (0 reserved for the `pcas` algorithm and 1-255 usable), 54-bit address annotated with user tag (64TB with 8-bit tag or 256GB with 16-bit tag)[10]. The tag is reserved for users to annotate arbitrary bits to pointer values for correctness [Harris 2001] or optimization [Chen et al. 2020]. We assume the ENCODE and DECODE functions respectively convert a ⟨parity, thread id, offset⟩ tuple to a location and vice versa.

As with the `chkpt` operation, `pcas` ensures atomicity by double buffering, storing two copies of a value and an annotated timestamp in its implementation of primitive memento. A 62-bit timestamp generated from `rdtscp` (sufficient for about 47 years without overflow) is annotated with a 1-bit

[9]For brevity, we defer the discussion on ABA prevention to §A.
[10]If 64 or more bits are necessary, a 118-bit integer supporting detectable CAS can be constructed from 128-bit machine words and double-word machine CAS operations.

parity and a 1-bit *success* flag, forming 8 bytes in total. We assume encodeT and decodeT convert a ⟨parity, success flag, timestamp⟩ tuple into an annotated timestamp and vice versa.

For helping, our framework tracks several timestamps in DRAM and PM. The *ts*.cas timestamp in DRAM records the parity-annotated timestamps of each thread's last CAS operation across crashes, while the global arrays $HELP[2][T]$ in PM record the timestamp of the last helping for each parity and thread, written by the helpers. Our framework maintains the invariant that the thread *ts*'s CAS was helped if *ts*.cas is less than $HELP[p][ts]$ for some appropriate parity p (see below).

The crash handler initializes *ts*.cas with the maximum timestamp checkpointed in `pcas` primitive mementos when a thread crashes, and uses *HELP* to calculate $t_{\max}$ for clock calibration when the system crashes (§4.1).

Load. We present our `pcas` implementation in Algorithm 5. As `pcas` acquires a lock by temporarily tagging parity, success flag, and thread id to the location value in PM, we also implement `pload` that helps the ongoing `pcas` to release the lock, ensuring it reads a value persisted in PM. Specifically, Load (L1) performs an architecture-provided plain load and invokes Help (see below for details on helping). As such, both operations are oblivious to tags: their input and output location values are tagged with zero.

CAS: Normal Execution. The `pcas` operation (L5) begins by identifying the stale and latest values in the memento (L9). It then performs two main tasks: (1) determining whether the CAS operation was completed or crashed while executing previously with the latest values in the memento, and if so, returning the previous result value (L12-27); (2) if not, executing an actual CAS operation (L28-52). For easier understanding, we describes the second task first.

The CAS operation ❶ tries to lock the location by performing a plain CAS to the new value annotated with the next parity ($\neg p_{\text{own}}$) and the thread id (*tid*, L30-34); ❷ if unsuccessful, it finishes the operation after updating *ts*.`time` and persisting the failure to the memento (L36-41); ❸ ensures the operation is committed by flushing the plain CAS (L43); and ❹ completes the operation after updating *ts*.`time` (L44) and *ts*.cas (for the next CAS operation) (L46), persisting the success to the memento (L48), attempting to unlock the location by atomically clearing annotations (L50), and (regardless of the result) ensuring the writes to the memento are flushed (L51).

CAS: Replay. To demonstrate that the execution of `pcas` is deterministically replayed, we first define the following events of a pre-crash execution. *Commit* is the flush of the first plain CAS at L30. Note that this event does not coincide with the flush instruction at L43, as a write can be voluntarily flushed before requested. *Checkpoint* is the flush of memento writes at L39 and L47. *Unlock* is the flush of the second plain CAS at L50. Based on the timing of a crash, the memory state that can be observed during post-crash execution can be categorized as follows:

($\lightning_1$) *Before commit*: the latest timestamp in the memento (t_{mmt}) is less than or equal to[11] the thread's last observed timestamp (*ts*.`time`).
($\lightning_2$) *Between commit and checkpoint*: t_{mmt} is still less than or equal to *ts*.`time`. The location (*loc*) can have one of two states: ($\lightning_{2a}$) *loc* is still locked by the thread; or ($\lightning_{2b}$) *loc* is not locked by the thread as it is unlocked by another thread's helping.
($\lightning_3$) *After checkpoint*: t_{mmt} is greater than *ts*.`time`.

The replay algorithm (L12-27) exhaustively covers all the crash cases mentioned above. After decoding the memento's annotated timestamp (L11), it compares t_{mmt} and *ts*.`time`. If t_{mmt} is greater than *ts*.`time` (corresponding to $\lightning_3$), the pre-crash execution is replayed: it updates *ts*.`time` and if `pcas` was successful, returns `true` and *old* (L14); otherwise returns `false` and the value stored

[11] If the memento function is within a loop, it is possible for the timestamp of the memento and the thread's last observed timestamp to be equal.

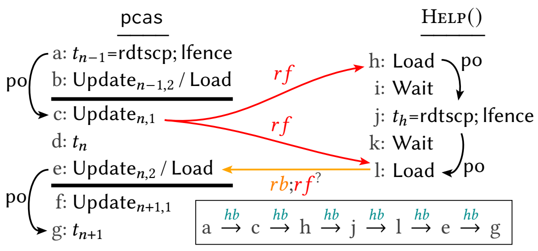
Algorithm 5 Load and detectable CAS for location values

```
1:  function pload($loc$)            ▷ for location values
2:      $cur \leftarrow \textsc{Load}_{\textsc{pln}}(loc)$
3:      return $\textsc{Help}(loc, cur)$
4:  end function
5:  function pcas($loc, old, new$, mid)
6:      $t_0 \leftarrow \textsc{Load}_{\textsc{pln}}(mmts[\text{mid}][0].\text{time})$
7:      $t_1 \leftarrow \textsc{Load}_{\textsc{pln}}(mmts[\text{mid}][1].\text{time})$
8:      $t_{\text{mmt}} \leftarrow (t_0 < t_1)\ ?\ t_1 : t_0$
9:      $(st, lt) \leftarrow (t_0 < t_1)\ ?\ (0,1) : (1,0)$
10:     $pst_{\text{mmt}} \leftarrow \textsc{Load}_{\textsc{pln}}(mmts[\text{mid}][lt].\text{time})$
11:     $(par_{\text{mmt}}, suc_{\text{mmt}}, t_{\text{mmt}}) \leftarrow \textsc{decode}\text{T}(pst_{\text{mmt}})$
12:     if $t_{\text{mmt}} > ts.\text{time}$ then
13:         $ts.\text{time} \leftarrow t_{\text{mmt}}$
14:         if $suc_{\text{mmt}}$ then return $(\texttt{true}, old)$
15:         $v_r \leftarrow \textsc{Load}_{\textsc{pln}}(mmts[\text{mid}][lt].\text{val})$
16:         return $(\texttt{false}, v_r)$
17:     end if
18:     $\_ \leftarrow \texttt{pload}(loc)$
19:     $(par_{\text{own}}, \_, t_{\text{own}}) \leftarrow \textsc{decode}\text{T}(ts.\text{cas})$
20:     $t_{\text{help}} \leftarrow \textsc{Load}_{\textsc{pln}}(HELP[\neg par_{\text{own}}][tid])$
21:     if $t_{\text{own}} < t_{\text{help}}$ then
22:         $ts.\text{time} \leftarrow t_{\text{help}}$
23:         $pst_{\text{suc}} \leftarrow \textsc{encode}\text{T}(\neg par_{\text{own}}, \texttt{true}, t_{\text{help}})$
24:         $\textsc{Store}_{\textsc{pln}}(mmts[\text{mid}][st].\text{time}, pst_{\text{suc}})$
25:         flushopt $mmts[\text{mid}][st].\text{time}$; sfence
26:         return $(\texttt{true}, old)$
27:     end if
28:     $old' \leftarrow \textsc{encode}(\textsc{Even}, \texttt{false}, 0, old)$
29:     $new' \leftarrow \textsc{encode}(\neg par_{\text{own}}, \texttt{false}, tid, new)$
30:     $r_1 \leftarrow \textsc{CAS}_{\textsc{pln}}(loc, old', new')$
31:     $t \leftarrow$ rdtscp; lfence
32:     if $r_1$ is $(\textsc{Err}\ cur)$ then
33:         $cur \leftarrow \textsc{Help}(loc, cur)$
34:         if $cur = old$ then goto 30
35:         $ts.\text{time} \leftarrow t$
36:         $pst_{\text{fail}} \leftarrow \textsc{encode}\text{T}(\textsc{Even}, \texttt{false}, t)$
37:         $ts.\text{cas} \leftarrow pst_{\text{fail}}$
38:         $\textsc{Store}_{\textsc{pln}}(mmts[\text{mid}][st].\text{val}, cur)$
39:         $\textsc{Store}_{\textsc{pln}}(mmts[\text{mid}][st].\text{time}, pst_{\text{fail}})$
40:         flushopt $mmts[\text{mid}][st]$; sfence
41:         return $(\texttt{false}, cur)$
42:     end if
43:     flushopt $loc$; sfence
44:     $ts.\text{time} \leftarrow t$
45:     $pst_{\text{suc}} \leftarrow \textsc{encode}\text{T}(\neg par_{\text{own}}, \texttt{true}, t)$
46:     $ts.\text{cas} \leftarrow pst_{\text{suc}}$
47:     $\textsc{Store}_{\textsc{pln}}(mmts[\text{mid}][st].\text{time}, pst_{\text{suc}})$
48:     flushopt $mmts[\text{mid}][st].\text{time}$
49:     $new'' \leftarrow \textsc{encode}(\textsc{Even}, \texttt{false}, 0, new)$
50:     $r_2 \leftarrow \textsc{CAS}_{\textsc{pln}}(loc, new', new'')$
51:     if $r_2$ is $\textsc{Err}$ then sfence
52:     return $(\texttt{true}, old)$
53: end function
54: function $\textsc{Help}(loc, old)$
55:     $(par_{\text{old}}, dsc_{\text{old}}, tid_{\text{old}}, o_{\text{old}}) \leftarrow \textsc{decode}(old)$
56:     if $tid_{\text{old}} = 0$ then return $o_{\text{old}}$
57:     wait($O_g$)
58:     $t \leftarrow$ rdtscp; lfence
59:     wait($O_g$)
60:     $cur \leftarrow \textsc{Load}_{\textsc{pln}}(loc)$
61:     if $old \neq cur$ then $old \leftarrow cur$; goto 55
62:     flushopt $loc$
63:     $r_{\text{dsc}} \leftarrow \textsc{RegisterDesc}(loc, old)$
64:     if $r_{\text{dsc}}$ is $(\textsc{Ok}\ cur)$ then
65:         $(par_{\text{old}}, \_, tid_{\text{old}}, o_{\text{old}}) \leftarrow \textsc{decode}(cur)$
66:     else
67:         $old \leftarrow \textsc{Load}_{\textsc{pln}}(loc)$; goto 55
68:     end if
69:     $t_{\text{help}} \leftarrow \textsc{Load}_{\textsc{pln}}(HELP[par_{\text{old}}][tid_{\text{old}}])$
70:     if $t \leq t_{\text{help}}$ then
71:         $old \leftarrow \textsc{Load}_{\textsc{pln}}(loc)$; goto 55
72:     end if
73:     $r \leftarrow \textsc{CAS}_{\textsc{pln}}(HELP[par_{\text{old}}][tid_{\text{old}}], t_{\text{help}}, t)$
74:     if $r$ is $\textsc{Err}$ then
75:         $old \leftarrow \textsc{Load}_{\textsc{pln}}(loc)$; goto 55
76:     end if
77:     flushopt $HELP[par_{\text{old}}][tid_{\text{old}}]$
78:     $old' \leftarrow \textsc{encode}(\textsc{Even}, \texttt{false}, 0, o_{\text{old}})$
79:     if $\textsc{CAS}_{\textsc{pln}}(loc, old, old')$ is $(\textsc{Err}\ \text{cur})$ then
80:         $old \leftarrow cur$; goto 55
81:     end if
82:     flushopt $loc$; sfence
83:     return $o_{\text{old}}$
84: end function
```

in the memento (L16). If t_{mmt} is less than or equal to *ts*.time, it helps the location's ongoing pcas if it exists (L18), which transitions the sub-case $\lightning_{2a}$ to $\lightning_{2b}$. To distinguish between cases $\lightning_1$ and $\lightning_{2b}$, the last timestamp increased by helper and the timestamp of the thread's last CAS operation should be compared. To this end, it decodes *ts*.cas, retrieves the parity and timestamp (t_{own}), and loads the helping timestamp (t_{help}) using the opposite parity (L19-20, see below for details of parity and timestamp on helping). If t_{help} is greater than t_{own} (corresponding to $\lightning_{2b}$), it detects (from the invariant of *ts*.cas and *HELP*) that the last CAS operation actually succeeded and finalizes the operation (L21-27). Otherwise (corresponding to $\lightning_1$), it proceeds to the normal execution (L28-52).

Helping. For lock-freedom, a thread may invoke Help(*loc*, *old*) (L54) for *loc*'s ongoing pcas operation to be flushed, unlock it and to return an unlocked (i.e. untagged) location value. It ❶ returns the given value *old* read from *loc* if is already unlocked (L56); ❷ waits for O_g, reads the current timestamp (t), and waits for O_g again to make t synchronized across other threads (L57-L59, see Observation 1); ❸ loads a value, say *cur*, from *loc* again, and if $old \neq cur$, then retries from L55 (L61); ❹ ensures the ongoing operation is committed by flushing *loc* (L62); ❺ registers the helping descriptor flag to prevent ABA[9] (L63-68); ❻ loads $HELP[par_{\text{old}}][tid_{\text{old}}]$, the last CAS help's timestamp for the parity and thread id annotated in *old*, and if it is bigger than t, retries the operation (L69-72); ❼ performs a plain CAS and flush on $HELP[par_{\text{old}}][tid_{\text{old}}]$ to atomically increase it to t, and if unsuccessful, retries the operation because the CAS has been already helped (L73-77); ❽ tries to unlock the location with a plain CAS and a flush, and if unsuccessful, retries the operation (L78-82); and ❾ returns the unlocked location (L83).

For deterministic replay, we show that Help() updates *HELP* for a re-execution of pcas to enter the branch at L21 if and only if the previous execution of pcas crashed between *commit* and *checkpoint* of success ($\lightning_2$). To this end, it is sufficient to prove the following.


pcas
Help()
a: t_{n-1}=rdtscp; lfence
b: Update$_{n-1,2}$ / Load
c: Update$_{n,1}$
d: t_n
e: Update$_{n,2}$ / Load
f: Update$_{n+1,1}$
g: t_{n+1}
po
po
h: Load
i: Wait
j: t_h=rdtscp; lfence
k: Wait
l: Load
po
po
rf
rf
$rb;rf^?$
$a \xrightarrow{hb} c \xrightarrow{hb} h \xrightarrow{hb} j \xrightarrow{hb} l \xrightarrow{hb} e \xrightarrow{hb} g$
Fig. 4. Synchronization of pcas() and Help()

Lemma 4.1. *Let p_n and t_n denote the parity and timestamp of tid's nth* pcas *invocation. The sequence $\{p_i\}$ then alternates between even and odd numbers, and the sequence $\{t_i\}$ is strictly increasing. Then $t_{n-1} < HELP[p_n][tid]$ if and only if either the nth or a later CAS with parity p_n was helped.*

Proof. Suppose a Help operation generates a timestamp t_h at L58 and tries to help the second plain CAS of thread *tid*'s n-th CAS invocation, as illustrated in Fig. 4. Here, we depict the plain CASes and timestamp generations of *tid*'s $(n-1)$st to $(n+1)$st CAS invocations, and loads and timestamp generations of a Help invocation, where Update$_{n,i}$ represents the ith plain CAS of *tid*'s nth CAS, and Load represents a load from a location. Then we have the following properties from Observation 1: **(1)** $t_{n-1} < t_h$ from $a \xrightarrow{po} c \xrightarrow{rf} h \xrightarrow{po} i \xrightarrow{po} j$; and **(2)** $t_h < t_{n+1}$ from $j \xrightarrow{po} k \xrightarrow{po} l \xrightarrow{rb;rf^?} e \xrightarrow{po} g$, where po is the program order; rf is the reads-from relation from each write to its readers; rb is the reads-before relation from each read to the later writes; $rb;rf^?$ is the reads-before relation possibly followed by a reads-from relation; and all relations constitute the *happens-before* relation hb in the x86-TSO memory model (see Owens et al. [2009] for more details).

Recall that Help persists Update$_{n,1}$ (L62), atomically increases $HELP[p_n][tid]$ to t_h (L73-L77), and helps Update$_{n,2}$ (L78-L82). If thread *tid*'s nth CAS was helped, then we have $t_{n-1} < t_h \leq HELP[p_n][tid]$ due to property **(1)**. Conversely, if $t_{n-1} < HELP[p_n][tid]$, then it cannot be the result of a help for $(n-2)$nd or earlier CASes or those with parity $\neg p_n$ due to property **(2)**. □

5 IMPLEMENTATION OF CONCURRENT DATA STRUCTURES

As primitive detectable operations, we implement ***Chkpt-mmt***: chkpt (§4.2); ***CAS-mmt***: pcas (§4.3); ***Indel-mmt***: *insertion/deletion* for atomic locations that performs fewer flushes than pcas. These primitives capture the essence of optimization in Friedman et al. [2018]; Li and Golab [2021]'s hand-tuned detectable Michael-Scott queues (MSQs) [Michael and Scott 1996] (see §B for details). Accordingly, we extend the core language to support additional primitive operations, including

Vol-mmt: a volatile location for cached values requiring no flushes (see §C.1 for details); and ***Comb-mmt***: an adaptation of Fatourou et al. [2022]'s general *combiner* for persistent DSs to our framework. While the original combiner is detectable, it only supports a *single* invocation of each operation by each thread, e.g. the following statements are not detectably recoverable:

```
1: $v_1 \leftarrow \textsc{Dequeue}(q)$;    $v_2 \leftarrow \textsc{Dequeue}(q)$;    $\textsc{Enqueue}(q, v_1 + v_2)$
```

If an execution crashes while performing DEQUEUE, we cannot detect whether it was for v_1 or v_2. In contrast, we distinguish the two invocations by distinct sub-mementos.

Using the primitives and our type system, we implement the following detectable, persistent DSs: ***List-mmt***: CAS-based lock-free linked-list; ***TreiberS-mmt***: CAS-based Treiber stack [Treiber 1986]; ***MSQ-mmt-O0***: CAS-based MSQ; ***MSQ-mmt-O1***: MSQ based on *Indel-mmt* and *Vol-mmt*; ***CombQ-mmt***: combining queue based on *Comb-mmt*; and ***Clevel-mmt***: CAS-based lock-free resizing hash table of Chen et al. [2020][12], which we optimize with an advanced type rule, LOOP-TRY (see §C.2 for details). Theorem 3.4 guarantees the detectability of these implementations. In addition, we implement ***MSQ-mmt-O2***: a variant of *MSQ-mmt-O1* with an *invariant-based optimization*, which reduces PM flushes based on the invariant that certain location values are always persisted (see §C.3 for details).

We implement safe memory reclamation for all DSs, but we defer the details to §D for space reasons. In doing so, we discover and fix a use-after-free bug in the MSQs of Friedman et al. [2018]; Li and Golab [2021] in case of crashes due to a lack of flush.

6 EVALUATION

We evaluate the detectable recoverability (§6.1) and performance (§6.2) of our detectable CAS, list, queues and hash table. We implement our framework and DSs in Rust nightly-2022-05-26 [Rust 2023] and build them with release mode. We use a machine running Ubuntu 20.04 and Linux 5.4 with dual-socket Intel Xeon Gold 6248R (3.0GHz, 24 cores, 48 threads) and an Intel Optane DCPMM (100 Series, 256GB). We pin all threads to a single socket to keep all DCPMM traffic within the same NUMA node. For brevity, we present only key results here; see §E for the full results.

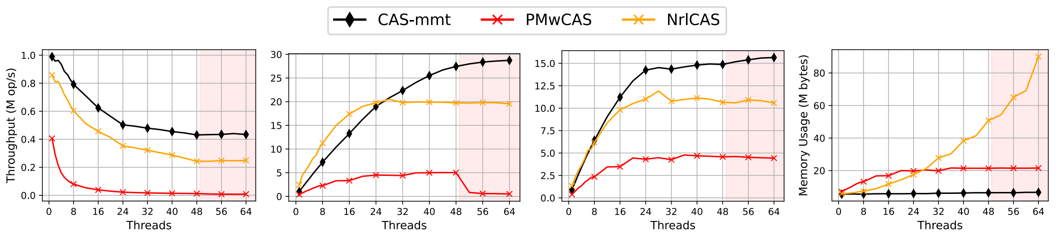
6.1 Detectability

We evaluate the detectability of two distinct crash scenarios: thread crashes and system crashes. Thread crashes present a more non-deterministic and challenging aspect to address in comparison to system crashes. Conversely, system crashes provide an opportunity to examine if data is accurately retained in persistent memory, thereby enabling the detection of missing flush bugs in weak persistency memory models [Cho et al. 2021].

To perform stress test under thread crashes, we randomly crash an arbitrary thread. To crash a specific thread, we use the `tgkill` system call to send the `SIGUSR1` signal to the thread and let its signal handler abort its execution. To the best of our knowledge, this is the first general stress test for thread crashes carried out for detectable, persistent DSs. For the integration test of CAS and each DS, **we observe no test failures for 100K runs with thread crashes**.

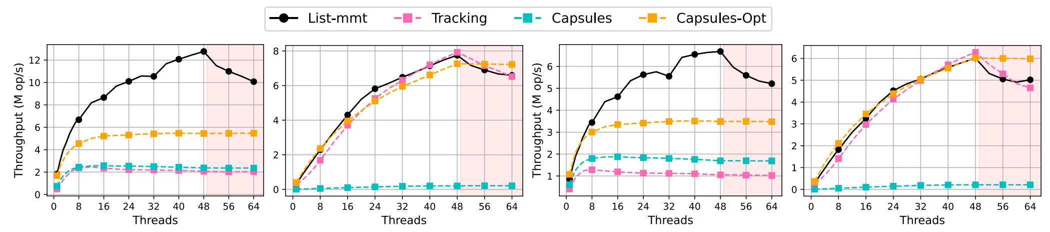
Provoking an actual system crash in a controlled and efficient manner is challenging within conventional systems. Instead, we perform stress test under *simulated* system crashes by running model-checking tools, Yashme [Gorjiara et al. 2022b] and PSan [Gorjiara et al. 2022a], in the "random" mode, which does not enumerate all possible executions and thus possibly fails to detect existing bugs. We use the random mode to avoid state explosion. For the integration test of CAS and each DS, **we observe no test failures for 1K runs with simulated system crashes.**

[12]We use a bug-fixed version due to Chen et al. [2022].



(a) Throughput (1 Location) (b) Throughput (1K Locations) (c) Throughput (1M Locations) (d) Memory Usage (1M Locations)

Fig. 5. Multi-threaded throughput of detectable CASes



(a) Read Intensive (20 Key Range) (b) Read Intensive (2K Key Range) (c) Update Intensive (20 Key Range) (d) Update Intensive (2K Key Range)

Fig. 6. Multi-threaded throughput of persistent lists

6.2 Performance

Unless specified otherwise, we measure the throughput for a varying number of threads: 1 to 8 and the multiples of 4 from 12 to 64; we report the average throughput of 5 runs, each for 10 seconds.

CAS. Fig. 5 presents the throughput and memory usage of our *CAS-mmt*; ***PMwCAS***: detectable multi-word CAS by Wang et al. [2018]; and ***NrlCAS***: detectable CAS by Attiya et al. [2018]. We reimplement *PMwCAS* in Rust to use the same allocator as the other CASes; we implement *NrlCAS* in Rust because its source code is not publicly available. Over-subscription over 48 threads is indicated as shaded regions. **(1)** When multiple threads perform CASes randomly on a varying number of locations (Fig. 5a, Fig. 5b, Fig. 5c), *CAS-mmt* exhibits higher throughput than the others for every thread count, except for *NrlCAS* with low thread counts for 1K locations. **(2)** When multiple threads perform CASes randomly on 1M locations (Fig. 5d), *NrlCAS* indeed consumes $O(T^2)$ PM locations, where T is the number of threads. **(3)** *PMwCAS* exhibits lower throughput than reported by Wang et al. [2018], because PM was not generally available at the time of writing and they experimented with DRAM. Also, *PMwCAS* generally exhibits lower throughput than single-word CASes because it supports multi-word CAS.

List. Fig. 6 illustrates the throughput of *List-mmt*; ***Capsule***: detectable linked-list by Ben-David et al. [2019]; and ***Capsule-Opt***: optimized detectable linked-list and ***Tracking***: detectable linked-list by Attiya et al. [2022]. We use the DS implementation and evaluation workloads of Attiya et al. [2022]: from a random initial list, read-intensive workloads perform inserts, deletes and finds for 15%, 15%, and 70% times; and update-intensive workloads perform them for 35%, 35%, and 30% times. **(1)** For small key ranges, *List-mmt* significantly outperforms the others thanks to fewer flushes to PM of timestamp-based replay (Fig. 6a, Fig. 6c); and **(2)** for large key ranges, all lists are saturated at almost the same performance because search dominates the cost (Fig. 6b, Fig. 6d).

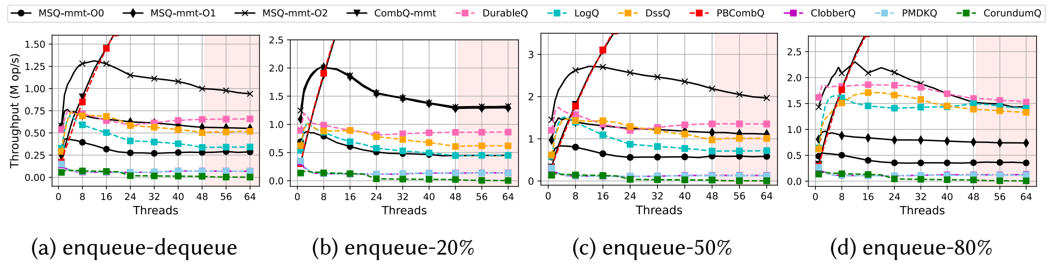


Fig. 7. Multi-threaded throughput of persistent queues

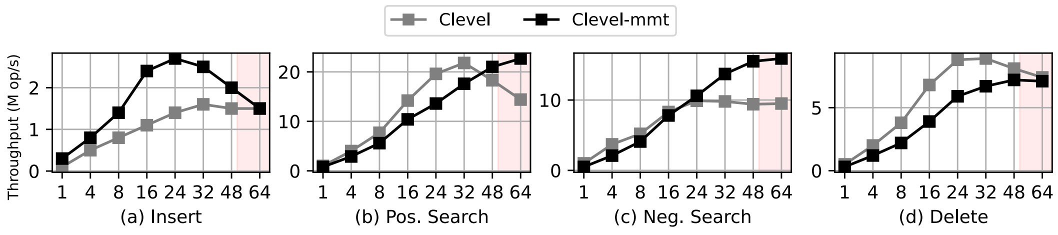


Fig. 8. Multi-threaded throughput of hash tables for uniform distributions

Queue. We compare the throughput of our queues; ***DurableQ***: undetectable durable MSQ by Friedman et al. [2018]; ***LogQ***: detectable MSQ by Friedman et al. [2018]; ***DssQ***: detectable MSQ by Li and Golab [2021]; ***PBcombQ***: detectable combining queue by Fatourou et al. [2022]; ***ClobberQ***: transaction-based queue in Clobber-NVM [Xu et al. 2021]; ***PMDKQ***: transaction-based queue in PMDK [Intel 2023b]; and ***CorundumQ***: transaction-based queue in Corundum [Hoseinzadeh and Swanson 2021]. We reimplement *DurableQ* and *LogQ* in Rust for a use-after-free bug (see §D); reimplement *PBcombQ* in Rust because it does not implement detectable recovery and uses a custom allocator; and implement *DssQ* in Rust because its source code is not publicly available.

Fig. 7 shows the throughput of queues for four workloads: ***enqueue-dequeue***: each operation enqueues an item and then immediately dequeues an item; ***enqueue-X%*** (with X=20, 50 or 80): each operation enqueues (or dequeues) an item for the probability of X%. We initialize the queues with 10M items to prevent excessive empty dequeues. **(1)** Transaction-based queues are noticeably slower than MSQs and combining queues. **(2)** Combining queues significantly outperform MSQs at high thread counts, in line with observations by Fatourou et al. [2022]. Thus, we cut the graphs to focus on MSQs rather than combining queues. **(3)** Although not shown in the graphs, it's worth noting that *CombQ-mmt* incurs a slight overhead over *PBcombQ*, especially for dequeue operations, because the latter saves a flush by assuming that a thread does not invoke an operation multiple times (see above). **(4)** *MSQ-mmt-O2* outperforms hand-tuned persistent MSQs with and without detectability thanks to fewer flushes to PM of timestamp-based deterministic replay (§2.2, see also §7). In addition, *DurableQ*'s dequeue incurs PM block allocation to store the return value. **(5)** *MSQ-mmt-O1* performs comparably with hand-tuned MSQs for dequeue-heavy workloads but not for enqueue-heavy workloads, because without an invariant-based optimization, its enqueue performs two plain CASes. **(6)** *MSQ-mmt-O0* is outperformed by hand-tuned MSQs due to its CAS-based dequeue flushing the head pointer and invalidating its cache line for every thread.

Hash Table. We compare the throughput of *Clevel-mmt* with the original, undetectable ***Clevel*** [Chen et al. 2020] using the PiBench benchmark [Lersch et al. 2019] specifically designed for PM hash tables. We employ the *Clevel* implementation and evaluation workloads from a PM hash table evaluation paper [Hu et al. 2021], which consists of seven workloads (insert, positive and negative search, delete, and write-heavy, balanced, and read-heavy) and two key distributions: uniform and skewed (80% of accesses target 20% of keys); see §E for full details.

Fig. 8 illustrates multi-threaded throughput of hash tables under uniform key distributions. The results for the skewed distribution are similar. **(1)** *Clevel-mmt* exhibits a slight overhead over *Clevel* for positive search queries because *Clevel-mmt*'s load operations checks if the location value is locked and it should help concurrent pcas (§4.3). **(2)** *Clevel-mmt* exhibits a noticeable overhead over *Clevel* for delete queries because *Clevel-mmt*'s delete operations perform two plain CASes for detectability. **(3)** *Clevel-mmt* outperforms *Clevel* for insert queries, and *Clevel* does not scale well over 24 threads. The main reason for this is that the PMDK allocator used by *Clevel* does not perform well for allocation and thread counts above the core count. While the comparison is not apple-to-apple, we can at least deduce that *Clevel-mmt*'s detectability introduces only modest overhead for most combinations of thread counts, workloads, and key distributions.

7 RELATED AND FUTURE WORK

Detectable Lock-Free DSs in PM. Attiya et al. [2022] propose transforming lock-free DRAM-based DSs into PM-based ones by persistently tracking an operation's progress and necessary completion information in its operation descriptor in PM. They assume each DS operation on the DS can be split into load-only gather and CAS-only update phases. However, this efficient approach is limited to specific operations that can be split in this manner and cannot handle complex operations with interleaved loads, CASes, or control constructs such as conditional branches and loops. Additionally, their approach performs a PM flush to reset an operation descriptor before reuse, while MEMENTO directly overwrites mementos without resetting, utilizing timestamps.

Ben-David et al. [2019] checkpoint program points and local variables to record an operation's progress and result. However, their approach has two limitations. First, it makes unrealistic system assumptions to recover the execution context from the checkpointed values correctly, such as the persistence of the OS page table and maintaining the same virtual address space upon recovery. These assumptions are not satisfied by Linux, which is typically used for PM deployments. Moreover, their method requires the number of each stack frame's persisted local variables to be less than a machine word's bitwidth to atomically update the validity of the local variables, limiting its applicability to complex operations in file systems and DBMS. Second, their approach must checkpoint program points around CASes and after branches, causing noticeable performance overhead, especially in write-heavy workloads, as shown in §6.

Friedman et al. [2018] and Li and Golab [2021] present detectable MSQs in PM, but both have a bug on reclamation (§D) and perform slower than our MSQ due to an additional flush (§6).

Rusanovsky et al. [2021] and Fatourou et al. [2022] present hand-tuned persistent combining DSs based on a general combiner. However, their DSs only support a single invocation for each operation (§5): their DSs use a *fixed* per-thread PM storage to track the progress of a thread's operation, and in our experience of implementing *CombQ-mmt*, storing the results of multiple invocations requires a sizeable restructuring of the algorithms. Furthermore, their methods require additional DS logic, requiring deep understanding: e.g. the combining queue of Fatourou et al. [2022] has extra synchronization that prevents dequeuing of elements that are enqueued but not yet persisted. By contrast, our type system applies to general programs with control constructs (Fig. 2) and automatically guarantees the detectability of well-typed programs (Theorem 3.1).

Undetectable Lock-Free DSs in PM. Friedman et al. [2018] present an undetectable lock-free MSQ in PM. Our detectable MSQ outperforms theirs because their dequeue operation allocates a PM block to store the return value (§6). Various hash tables [Chen et al. 2020; Lee et al. 2019; Lu et al. 2020; Nam et al. 2019; Zuo et al. 2018; Zuriel et al. 2019] and trees [Arulraj et al. 2018; Kim et al.

2021b] in PM have been proposed in the literature. In this paper, we convert the Clevel [Chen et al. 2020] hash table to a detectable one as a case study because it is lock-free. Converting the others to detectable DSs is an interesting direction for future work.

Transformation of DSs from DRAM to PM. Izraelevitz et al. [2016] present a universal construction of lock-free DSs in PM, but the constructed DSs are generally slow [Friedman et al. 2020, 2021]. Lee et al. [2019] propose a *RECIPE* to convert indexes from DRAM to PM and Kim et al. [2021b] propose the *Packed Asynchronous Concurrency* guideline to construct high-performance persistent DSs in PM, but their approaches are abstract, high-level, and not immediately applicable to DSs in general. By contrast, our rules of composition provide a more concrete guideline at the code level.

NVTraverse [Friedman et al. 2020] is a systematic transformation of persistent DSs, exploiting an observation that most operations comprise two phases: read-only traversal (which does not require flushes) and critical modification. Mirror [Friedman et al. 2021] is a more general and efficient transformation that replicates DSs in PM and DRAM, significantly improving read performance. FliT [Wei et al. 2022] is a persistent DS library based on a transformation utilizing dirty cache line tracking. However, none of these works support the transformation of detectable DSs.

Detectability. Attiya et al. [2022, 2018]; Friedman et al. [2018]; Li and Golab [2021] define detectability as *thread*'s ability to detect the DS operation's progress of a pre-crash execution and resume thereafter. We formalize this property as a deterministic replay of thread executions (Definition 3.2) and instead define detectability as failure transparency of *machine*'s behaviours under crashes, and generally prove the detectability of well-typed programs (Theorem 3.1).

PM Platforms. Memento applies not only to Intel Optane persistent memory [Intel 2019] (with and without eADR [Intel 2021]) but also to other PM platforms, such as Samsung's Memory-Semantic SSD (MS-SSD) [Samsung 2023], because they all provide the following features that Memento relies on: direct access via mmap and fine-grained data transfer. Intel's PMDK [Intel 2023b] maps persistent memory to virtual memory via mmap to support direct memory access [Intel 2023a], while Samsung's SMDK supports the CXL.mem interface [Samsung 2022] that serves the same purpose. Furthermore, both Intel Optane persistent memory and Samsung MS-SSD's CXL.mem interface transfer data at the cache line granularity [Blankenship 2020].

Future Work. **(1)** We will design larger objects (e.g. file systems and storage engines) in Memento. **(2)** In doing so, we will adapt existing hand-tuned detectable concurrent DSs and persistent transactional memory (PTM) systems [Krishnan et al. 2020; Memaripour et al. 2017] to Memento to compose them into larger objects. **(3)** We will formalize our type system and verify the detectability of well-typed programs in logics for PM [Raad et al. 2020; Vindum and Birkedal 2022]. **(4)** We will reason about the invariant-based optimizations to verify *MSQ-mmt-O2* by composing the type-based automatic verification of *MSQ-mmt-O1* and manual verification of the invariant-based optimization.

ACKNOWLEDGMENTS

We thank the PLDI 2023 reviewers and Minseong Jang for their valuable feedback. Kyeongmin Cho, Seungmin Jeon, and Jeehoon Kang are partly supported by Samsung Electronics Co., Ltd (No. IO220407-09486-01), and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea Government (MSIT) (No. 2018-0-00503, Researches on next generation memory-centric computing system architecture). Azalea Raad is supported by a UKRI Future Leaders Fellowship under grant number MR/V024299/1.

8 DATA AVAILABILITY STATEMENT

The accompanying artifact [Cho et al. 2023] (also available at https://github.com/kaist-cp/memento) provides the implementation and the experimental results. The artifact is structured as follows.

- `/memento/src/`: the implementation of the Memento framework and its primitives (§4).
- `/memento/src/`: the implementation of detectably persistent data structures (§5).
- `/memento/evaluation/`: the experiment script for correctness and performance (§6).
- `/evaluation_data/`: the complete experimental results (§6).

Please refer to the artifact's `README.md` for detailed instructions on how to reproduce the results.

REFERENCES

Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (jan 2018), 553–565.

Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. 2019. Tracking in Order to Recover: Detectable Recovery of Lock-Free Data Structures. https://doi.org/10.48550/ARXIV.1905.13600

Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. 2022. Detectable Recovery of Lock-Free Data Structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 262–277. https://doi.org/10.1145/3503221.3508444

Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing* (Egham, United Kingdom) *(PODC '18)*. Association for Computing Machinery, New York, NY, USA, 7–16. https://doi.org/10.1145/3212734.3212753

Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-Free Concurrency on Faulty Persistent Memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures* (Phoenix, AZ, USA) *(SPAA '19)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/3323165.3323187

Jim Blandy. 2022. Comparison of Rust async and Linux thread context switch time and memory use. https://github.com/jimblandy/context-switch

Robert Blankenship. 2020. CXL 1.1 Protocol Extensions: Review of the Cache and Memory Protocols in CXL. https://www.snia.org/educational-library/cxl-11-protocol-extensions-review-cache-and-memory-protocols-cxl-2020

Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management* (London, UK) *(ISMM 2020)*. Association for Computing Machinery, New York, NY, USA, 60–73. https://doi.org/10.1145/3381898.3397212

Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, Marcos K. Aguilera and Gala Yadgar (Eds.). USENIX Association, 81–95. https://www.usenix.org/conference/fast21/presentation/chen-youmin

Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 799–812. https://www.usenix.org/conference/atc20/presentation/chen

Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. 2022. Efficiently Detecting Concurrency Bugs in Persistent Memory Programs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 873–887. https://doi.org/10.1145/3503222.3507755

Kyeongmin Cho, Seungmin Jeon, Azalea Raad, and Jeehoon Kang. 2023. Artifact for Article "Memento: A Framework for Detectable Recoverability in Persistent Memory". https://doi.org/10.5281/zenodo.7811928

Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Revamping Hardware Persistency Models: View-Based and Axiomatic Persistency Models for Intel-X86 and Armv8. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 16–31. https://doi.org/10.1145/3453483.3454027

Jeongdong Choe. 2022. Review and Things to Know: Flash Memory Summit 2022. *TechInsights* (August 2022). https://www.techinsights.com/blog/review-and-things-know-flash-memory-summit-2022

Crossbeam. 2022. Crossbeam. https://github.com/crossbeam-rs/crossbeam

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 373–385.

Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 325–334. https://doi.org/10.1145/1989493.1989549

Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/2145816.2145849

Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios Kosmas. 2022. The Performance Power of Software Combining in Persistence. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 337–352. https://doi.org/10.1145/3503221.3508426

Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. *Mirror: Making Lock-Free Data Structures Persistent.* Association for Computing Machinery, New York, NY, USA, 1218–1232. https://doi.org/10.1145/3453483.3454105

James R. Goodman, Mary K. Vernon, and Philip J. Woest. 1989. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS III)*. Association for Computing Machinery, New York, NY, USA, 64–75. https://doi.org/10.1145/70082.68188

Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. 2022a. Checking Robustness to Weak Persistency Models. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 490–505. https://doi.org/10.1145/3519939.3523723

Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2022b. Yashme: Detecting Persistency Races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 830–845. https://doi.org/10.1145/3503222.3507766

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/3445814.3446710

Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *Proc. VLDB Endow.* 14, 5 (jan 2021), 785–798. https://doi.org/10.14778/3446095.3446101

Intel. 2019. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

Intel. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

Intel. 2022. Intel 64 and IA-32 Architectures Software Developer's Manual (Combined Volumes). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html Order Number: 325462-076US.

Intel. 2023a. The libpmem2 library. https://pmem.io/pmdk/libpmem2/

Intel. 2023b. Persistent Memory Programming. https://pmem.io/pmdk/

Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.

Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: A Hugepage-Aware File System for Persistent Memory That Ages Gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 804–818. https://doi.org/10.1145/3477132.3483567

Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 34, 15 pages. https://doi.org/10.1145/3190508.3190510

Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021a. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 756–771. https://doi.org/10.1145/3477132.3483565

Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021b. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 424–439. https://doi.org/10.1145/3477132.3483589

R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 335–349. https://doi.org/10.1145/3373376.3378483

Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (dec 2019), 574–587. https://doi.org/10.14778/3372716.3372728

Nan Li and Wojciech Golab. 2021. Brief Announcement: Detectable Sequential Specifications for Recoverable Shared Objects. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 557–560. https://doi.org/10.1145/3465084.3467943

Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

Paul E McKenney. 2005. Memory ordering in modern microprocessors, part I. *Linux Journal* 2005, 136 (2005), 2.

John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. 2015. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.* 8, 13 (sep 2015), 2134–2145. https://doi.org/10.14778/2831360.2831367

Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 499–512. https://doi.org/10.1145/3064176.3064215

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, USA) *(PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

Microsoft. 2023. High context switch rate. https://learn.microsoft.com/en-us/gaming/gdk/_content/gc/system/overviews/finding-threading-issues/high-context-switches

Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better X86 Memory Model: X86-TSO. In *TPHOL*.

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020. Persistent Owicki-Gries Reasoning: A Program Logic for Reasoning about Persistent Programs on Intel-X86. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 151 (nov 2020), 28 pages.

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (dec 2019), 31 pages. https://doi.org/10.1145/3371079

Ganesan Ramalingam and Kapil Vaswani. 2013. Fault Tolerance via Idempotence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 249–262. https://doi.org/10.1145/2429069.2429100

Matan Rusanovsky, Hagit Attiya, Ohad Ben-Baruch, Tom Gerby, Danny Hendler, and Pedro Ramalhete. 2021. Flat-Combining-Based Persistent Data Structures for Non-Volatile Memory. arXiv:2012.12868 [cs.DC]

Rust. 2023. Rust. https://www.rust-lang.org/

Samsung. 2022. API list of Scalable Memory Development Kit (SMDK). https://github.com/OpenMPDK/SMDK/wiki/5.-Plugin#api-list

Samsung. 2023. Memory-Semantic SSD. https://samsungmsl.com/ms-ssd/

Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 501–516.

Ori Shalev and Nir Shavit. 2006. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *J. ACM* 53, 3 (may 2006), 379–405. https://doi.org/10.1145/1147954.1147958

Shahar Timnat and Erez Petrank. 2014. A Practical Wait-Free Simulation for Lock-Free Data Structures. *SIGPLAN Not.* 49, 8 (feb 2014), 357–368. https://doi.org/10.1145/2692916.2555261

R.K. Treiber. 1986. *Systems Programming: Coping with Parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center. https://books.google.co.kr/books?id=YQg3HAAACAAJ

Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building Blocks for Persistent Memory: How to Get the Most out of Your New Memory? *The VLDB Journal* 29, 6 (nov 2020), 1223–1241. https://doi.org/10.1007/s00778-020-00622-9

Simon Friis Vindum and Lars Birkedal. 2022. Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory. https://cs.au.dk/~vindum/res/spirea.pdf

Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. 2021. *Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka.* Association for Computing Machinery, New York, NY, USA, 2602–2613.

Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 461–472. https://doi.org/10.1109/ICDE.2018.00049

Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2022. FliT: A Library for Simple and Efficient Persistent Algorithms. In *Proceedings of the The 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, South Korea) *(PPoPP 2022)*.

Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, Angela Demke Brown and Florentina I. Popovici (Eds.). USENIX Association, 323–338.

Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-Execute More. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 346–359.

Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. 2021. Octopus$^+$: An RDMA-Enabled Distributed Persistent Memory File System. *ACM Trans. Storage* 17, 3 (2021), 19:1–19:25. https://doi.org/10.1145/3448418

Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo

Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 128 (Oct. 2019), 26 pages. https://doi.org/10.1145/3360554

Received 2022-11-10; accepted 2023-03-31