Compiler-Assisted Crash Consistency for PMEM

Yun Joon Soh
University of California San Diego
San Diego, USA
yjsoh@ucsd.edu

Sihang Liu
University of Waterloo
Waterloo, Canada
sihangliu@uwaterloo.ca

Steven Swanson
University of California San Diego
San Diego, USA
sjswanson@ucsd.edu

Jishen Zhao
University of California San Diego
San Diego, USA
jzhao@ucsd.edu

Abstract

Writing crash-consistent programs for memory-semantic storage such as persistent memory (PMEM) is error-prone and cumbersome. Programmers must implement both the main logic and the recovery logic to ensure data consistency after unexpected power failures. Prior work has reduced this burden using compiler-assisted logging techniques to enforce crash consistency. However, these techniques often apply persistence uniformly, limiting support for diverse programming models and incurring high logging overhead.

We present **SSAPP (Statically and Systematically Automated Persistence is Possible)**, a compiler extension that transparently adds crash consistency to the main logic and automatically generates tailored recovery code. SSAPP persists transient state with low overhead during main logic execution and makes principled resumption decisions during post-failure recovery. Based on these decisions, the generated recovery code correctly completes the interrupted operation. This design supports a broader range of programming models — including lock-free data structures — while reducing crash consistency overhead.

We evaluate SSAPP on transactional benchmarks, lock-based, and lock-free data structures. With minimal developer effort, SSAPP converts volatile lock-free data structures into crash-consistent ones, achieving performance comparable to Mirror, a hand-optimized persistent data structure library. SSAPP also outperforms Clobber-NVM, a prior compiler-based PMEM system, achieving 1.8× higher throughput.

CCS Concepts: • **Software and its engineering** → **Software fault tolerance**; *Automatic programming*; *Runtime environments*; Consistency.

Keywords: Persistent Memory, Crash Consistency, Compiler-based Transformation, Automatic Recovery

ISMM '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1610-2/25/06
https://doi.org/10.1145/3735950.3735955

ACM Reference Format:
Yun Joon Soh, Sihang Liu, Steven Swanson, and Jishen Zhao. 2025. Compiler-Assisted Crash Consistency for PMEM. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management (ISMM '25), June 17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3735950.3735955

1 Introduction

Persistent memory (PMEM) technologies promise to bridge the gap between memory and storage by offering high capacity, non-volatility, DRAM-like performance with byte addressability. Despite the benefits, programming for PMEM introduces unique challenges that differ from traditional memory or block-based storage. In particular, ensuring crash consistency — allowing a program to recover to a consistent state after an unexpected power failure — requires careful reasoning about hardware memory models, cache behavior, and recovery mechanisms.

We assume the following programming and failure model: CPU caches are **non-volatile**, meaning that data in caches is preserved across power failures. This assumption simplifies the persistence model: explicit cache-line flushes are unnecessary, and only memory ordering (via fence instructions) must be enforced for correct crash consistency.

Developing crash-consistent programs even under this simplified model remains difficult and error-prone. A key challenge is maintaining *failure-atomicity*, where a group of relevant memory updates must either all be reflected in storage or none at all. Developers must manually reason about update ordering and atomicity, inserting memory barriers at appropriate code boundaries. Overuse of such instructions degrades performance, while underuse leads to inconsistency.

Furthermore, programming complexity is exacerbated in multi-threaded or lock-free programs, where recovery logic becomes entangled with the main logic of the program, demanding rigorous planning and testing. To make matters worse, verifying the crash consistency of a program is an open research problem, as many works continuously uncover new types of bugs [6, 10, 11, 18, 26, 32, 35, 43].

Prior work has explored various approaches, including persistent memory (PMEM) programming libraries [2, 4, 15, 17, 29, 47], static code analysis for conservative flush and fence insertion [42], hardware support for persistent cache emulation, and compiler extensions [8, 19, 31, 50]. Each approach has limitations: libraries reduce programming complexity but restrict flexibility; persistent caches still require manual recovery code; and naively inserting flush and fence instructions hurts runtime performance.

Recovery-Via-Resumption (RVR) [19, 31, 50], is a compiler-based technique, which assumes that failure-atomic sections (FASEs) are clearly delineated, either via locks or programmer annotations. Recovery is achieved by resuming execution from the logged program counter and executing until completion. However, these models do not generalize to lock-free or fine-grained programs, where atomicity is not explicitly marked and operations often involve read-modify-write (RMW) sequences. For example, naively re-executing a fetch-and-add after a crash (because the program counter indicates that it was the failure point) may result in applying the update twice. We elaborate on these limitations and explain how RVR negatively impacts the performance of hand-optimized programs in Section 2.3.3.

This paper presents **SSAPP (Statically and Systematically Automated Persistence is Possible)**, a compiler extension that generalizes RVR to support a broader range of programs, including lock-free data structures. SSAPP statically analyzes the input code and partitions it into read-only, write-only, or RMW code regions. Then, SSAPP systematically transforms it into a crash-consistent main logic and a tailored recovery code. Unlike existing approaches, SSAPP inserts more precise decision logic into automatically generated recovery code, allowing it to either re-execute or **skip the crashed region**, such as an already executed RMW region. To clarify the limitation, SSAPP depends on several assumptions as detailed in Section 3.1.2. The key programmer responsibility is to define each failure-atomic region as a single function.

The pre-failure performance and recovery correctness depend on low-overhead techniques that answer the following two questions: (Q1) which code region was being executed at the moment of a crash, and (Q2) whether the recovery should re-execute or skip the crashed region. We propose three techniques to answer these questions: (T1) *Crash Buoy (CB)*, a program progress tracker, (T2) *Load-to-Persist (L2P)*, a lightweight load atomicity tracking technique, and (T3) *Tornbit for Store Atomicity (TSA)*, a versioning bit embedded in metadata to verify atomicity of stores and decide whether to re-execute from or skip over the crashed region.

These mechanisms work together to answer (Q1) and (Q2) in the following way. Similar to previous RVR techniques that persist the program counter for each load or store instruction, SSAPP persists the code region ID to a dedicated per-thread storage location called **Crash Buoy (CB)**. The per-thread CB enables multi-threading support, where each thread may be working on a different region of the same code. Instead of logging the data to be overwritten as in an undo log, SSAPP ensures that the loaded values are consistent via **Load-to-Persist (L2P)**. To ensure correct resumption decisions, SSAPP adds a single-bit versioning mechanism called **Tornbit for Store Atomicity (TSA)** to each metadata updated by SSAPP. If the TSA values from CB and L2P match within the crashed region, the region has been atomically executed before the failure and could be skipped. The proposed technique is CPU-cache-friendly and requires only half the number of memory fence instructions compared to logging, resulting in higher performance.

We evaluate SSAPP for various applications with persistent cache assumption, including lock-free/lock-based data structures and transaction benchmarks against Mirror [7], Clobber-NVM [50], hand-optimized versions, and PMDK [17]. Overall, SSAPP achieves 1.8× higher throughput than Clobber-NVM [50], a compiler-based recovery-via-resumption approach. The performance benefit stems from two key factors: the proposed techniques are lightweight, cache-friendly, and relaxed ordering requirements within each code region. CB, L2P, and TSA involve bit operations and store updates to per-thread, per-region, preallocated memory. Even when the same code region executes repeatedly (e.g., in loops), SSAPP reuses preallocated metadata storage. This reuse improves cache locality, as the metadata is often already cached. Furthermore, the versioning information is embedded in the metadata, eliminating the need for explicit ordering between version updates and their corresponding data writes.

In summary, this paper makes the following contributions.

- Instead of requiring programmers to manually write, test, and debug crash-consistent code for PMEM, SSAPP transparently provides crash consistency, improving programmability and reducing developer burden.
- We implemented SSAPP, a compiler extension that adds crash consistency to the input program and automatically generates a matching post-failure recovery code.
- SSAPP supports a wider range of programs, including lock-free data structures, and exhibits comparable performance with the hand-optimized persistent lock-free library, Mirror, without burdening the programmer to add crash consistency manually.
- SSAPP improves 1.8× throughput over Clobber-NVM.

We organize the paper as follows: We introduce programming models for PMEM and summarize existing works' limitations in Section 2. Then, we address the challenges and the critical design choices in Section 3. After elaborating on the implementation (Section 4), we report the performance benefit, scalability of SSAPP, and empirical correctness validation (Section 5). We discuss related works in Section 6 and conclude in Section 7.

2 Background

We outline the fundamental building blocks and survey existing solutions for writing crash-consistent persistent memory (PMEM) programs.

2.1 Crash-Consistent Programming Primitives

PMEM exposes memory directly to a process's virtual address space via DAX-`mmap()`. This bypasses the traditional kernel I/O stacks, enabling programs to perform fine-grained updates that leverage PMEM's low latency and non-volatility. Once mapped, a program performs updates using standard load and store instructions. However, since modern architectures may reorder these instructions for performance, a crash can leave memory in an inconsistent and unrecoverable state.

To address this, programmers insert cache-line flush and memory barrier instructions to align persistence order with program execution order. The former flushes dirty cache lines to backing storage, while the latter stalls program execution until preceding memory instructions retire. While these primitives are foundational for crash consistency, they are easily misused. Redundant flushes to the same cache line degrade performance, and excessive fence instructions add unnecessary latency. Conversely, missing a flush or fence may violate memory ordering guarantees, resulting in inconsistencies after a crash.

2.2 Persistent CPU Cache

Prior work [19] assumes the presence of non-volatile CPU caches, backed by capacitors, allowing dirty cache lines to be flushed to PMEM upon power failure. Despite the discontinuation of Intel Optane PMEM, several emerging platforms such as extended Asynchronous DRAM Refresh (eADR) exist along with systems such as memory-semantic SSD and CXL shared memory continue to offer partial failure resilience [51, 56].

However, eADR is not a silver bullet for crash consistency. It does not eliminate crash consistency challenges due to (1) the use of non-temporal store instructions and (2) the lack of a universal definition of a consistent state. Even with eADR, programmers must carefully insert fence instructions when using non-temporal stores (weakly-ordered, cache-bypassing store instructions), considering the associated performance overhead. Furthermore, eADR guarantees persistence only up to a point; applications still need their own definitions of consistency. For instance, lock-free data structures require crash consistency logic, as an interrupted operation such as rebalancing a tree can leave memory in a logically inconsistent state.

2.3 Programming Models and Tools

To ease crash consistency, industry and academia have introduced various programming models such as crash-consistent transactions, Failure-Atomic SEctions (FASE), and recovery-via-resumption (RVR).

2.3.1 Crash-Consistent Transactions. Crash-consistent transactions typically encapsulate a sequence of operations within start/end markers (e.g., API calls), ensuring all updates in-between become visible or none do. A widely adopted technique for crash-consistent transactional memory is write-ahead logging, which persists auxiliary recovery data called logs. It ensures atomicity by persisting either the original version (undo-log) or the updated version (redo-log) of each data object. Recovery involves rolling back partial transactions or reapplying the committed ones. From an execution standpoint, logging requires memory fences to guarantee that log entries are persisted before in-place updates occur, ensuring correctness. This incurs high overhead, and many previous works attempt to reduce it through various solutions, including hardware-based solutions [22, 36, 44, 48, 57], software approaches [8, 37–39] and specialized data structures where relative offsets encode sufficient recovery information [41].

2.3.2 Failure-Atomic Section (FASE). FASEs are code regions identified by the outermost locks [2]. Chakrabarti *et al.* observed that a data-race-free program with lock-based concurrency controls naturally creates code regions where updates are expected to be failure-atomic. Within each region, all updates should persist together or not at all. Subsequent work extended this model using relaxed memory barriers [5], or compiler-based recovery mechanisms [19, 31, 50].

2.3.3 Recovery-Via-Resumption. Even with persistent caches, traditional logging incurs overhead proportional to the modified data [19]. The authors proposed an alternative approach to traditional undo/redo logging: a minimalistic logging before each store instruction, consisting of the program counter, store address, and value. Upon recovery, JUSTDO loads the latest program counter and resumes the execution until completion (thus referred to as recovery-via-resumption). This method outperforms traditional logging, despite the increased logging frequency. Although their solution inspired various follow-up works focusing on transient CPU cache [21, 31, 50, 53], they were limited by restrictive assumptions such as prohibiting transient state (e.g., data on DRAM, variable promoted to register).

Follow-up works, such as iDO [31] and Clobber-NVM [50], addressed the limitation of JUSTDO and reduced the crash consistency overhead. iDO identified that a set of stores (those within an idempotent region) can be re-executed without side effects and, therefore, does not require prior logging. Clobber-NVM further reduced the logging size by identifying clobber writes — writes that overwrite inputs — and logging them just before execution. However, these works relied on

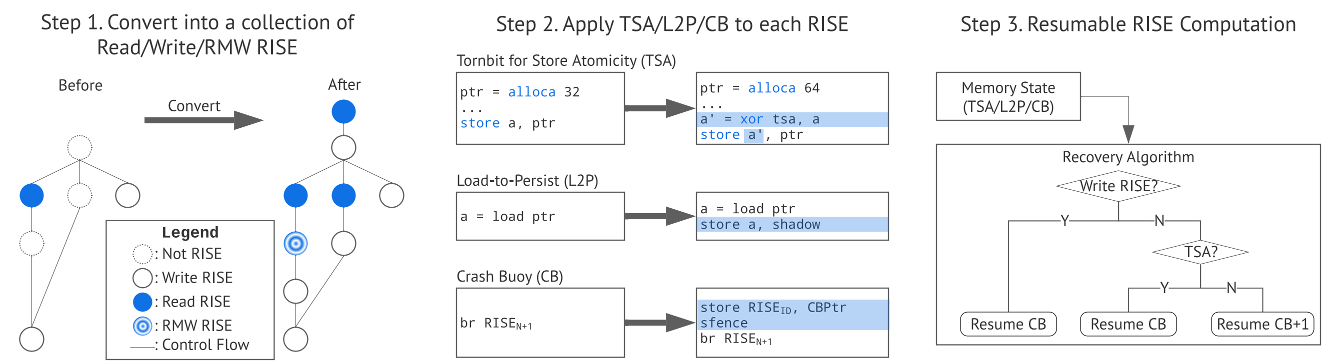


Figure 1. Key Design Components. The SSAPP pipeline begins by converting input control flow into a collection of read/write/RMW RISE units (Section 3.2). Tornbit for Store Atomicity (TSA) is applied by embedding version bits into memory writes (Section 3.4.1). Load-to-Persist (L2P) ensures read values are persisted with torn-bits (Section 3.4.2). Each RISE stores execution progress in a per-thread Crash Buoy (CB) to approximate failure points (Section 3.4.3). Based on the memory state, the recovery logic determines the correct resumable RISE (Section 3.5).

a FASE-based programming model, and applying them to a non-FASE model could lead to suboptimal performance or consistency issues.

2.4 Motivation

An overlooked strength of RVR is its potential to serve as a foundation for transparently providing crash consistency and reducing the burden of writing recovery code. Furthermore, RVR always completes an initiated FASE, allowing the programmer to regard it as committed as soon as its initiation phase is complete [50]. From this observation, we propose a systematic, compiler-assisted approach that automates the recovery generation for a broader range of programming models with better performance.

We propose **SSAPP**, a systematic approach that generalizes RVR beyond FASEs. SSAPP rewrites pre-failure code into a series of failure-Reentrant and Idempotent SEction (RISE) and instruments them with progress-tracking and atomicity-checking logic. On recovery, SSAPP decides whether to skip or re-execute the crashed RISE based on the tracked state.

SSAPP requires minimal programmer input (function-level atomicity boundaries) and supports various programming models, including lock-free code. The key enabler is that SSAPP generates a tailored recovery algorithm that uses persisted metadata to make resumption decisions based on the atomicity of prior memory updates. With a delicate resumption decision, SSAPP transparently adds crash consistency to various programming models, including lock-free data structures.

3 Design

Writing crash-consistent programs for persistent memory (PMEM) is error-prone and cumbersome because developers must explicitly manage both the main logic and the recovery logic. To address these challenges, we propose **SSAPP**, claiming that **Statically and Systematically Automated Persistence is Possible**. SSAPP is a novel compiler extension that transparently ensures crash consistency with three key components: (1) TSA, a lightweight, atomicity-checking mechanism embedded directly into memory operations; (2) L2P, an efficient persistence mechanism that eliminates the overhead of expensive logging; and (3) Crash Buoy, a low-overhead approximate progress tracker enabling correct resumption without enforcing expensive memory ordering constraints. Table 1 defines key terms and their respective roles within SSAPP.

Unlike prior works [31, 50], which required memory barriers between logging and data updates, SSAPP embeds single-bit versioning information directly into log entries, eliminating the need for an additional barrier. Using this embedded version information, SSAPP intelligently determines whether a set of memory accesses was atomically executed before a sudden failure, avoiding naive resumption from previously recorded program progress. This approach yields two key benefits: (1) broader support for diverse programming models beyond naive Recovery-Via-Resumption (RVR), and (2) reduced overhead by eliminating memory barrier with a lightweight, cache-friendly alternative techniques.

3.1 Challenge and Assumptions

Designing a universal solution for automatic recovery generation imposes several technical challenges. Although we target a generic solution suitable for a wide range of programming models, not all programs are appropriate candidates for SSAPP. Here, we articulate these technical challenges and clearly outline the programmer's responsibilities.

3.1.1 Design Challenges. The core challenge is ensuring that each basic block (BB) can be safely re-executed without

Table 1. Key Terminologies.

Term	Definition	Purpose
RISE	Reentrant & Idempotent Code Section	Ensure safe re-execution
CBID	Crash Buoy ID (coarse-grain PC)	Approximate tracking of crash location
TSA	Torn-bit for Store Atomicity	Determine if RISE execution is atomic
L2P	Load-to-persist	Persist transient state efficiently
WAR	Write-After-Read	Critical anti-idempotency pattern
Basic Block (BB)	A straight-line sequence of instructions with a single entry and a single exit	Basic unit of compiler pass operation
Atomicity	Ensures that an instruction sequence executes either completely or not at all	Maintains data consistency

side effects and that subsequent BBs do not introduce unintended behavior. This requires several tasks: (1) ensuring idempotency of each BB, (2) persisting the transient state efficiently for failure reentrancy, (3) embedding minimal information to verify whether a RISE executed atomically prior to a failure and (4) ensuring a correct resumption for crash consistency.

A resumption decision is correct if the transient data required for resumption is available in a consistent state, and resumption does not result in an incorrect re-execution. For example, consider a simple increment operation, `i++`. The initial value of `i` must be available after a crash, and the recovery thread must ensure it executes only once, either by the pre-failure thread or the recovery thread.

Unlike prior works, which relied on alias analysis to find idempotent code regions, we observe that loading an already updated value is a primary source of incorrect resumption. Therefore, SSAPP separates sequences of load instructions from other instructions by inserting unconditional branch instructions. For example, the `i++`, which is a sequence of load-add-store, becomes load-**branch**-add-store. This greedy strategy allows SSAPP to avoid conservative alias analysis, which typically results in unnecessary logging overhead.

When persisting transient data (e.g., stack variable, current RISE ID), naive logging or checkpointing techniques incur significant performance overhead because data must be persisted prior to updates. Instead, SSAPP leverages the fact that all virtual registers (variables in LLVM) must be defined before their use due to the Single Static Assignment (SSA) requirement. LLVM uses `alloca` instruction to allocate space on the stack when defining a virtual register. SSAPP replaces these with function calls to the PMEM allocator so that the subsequent memory accesses target PMEM. Given that Extended Asynchronous DRAM Refresh (eADR) guarantees persistence for cached data, SSAPP avoids explicitly flushing caches, thus significantly reducing transient data persistence overhead.

To verify whether a RISE executed atomically, SSAPP embeds minimal information into each memory access instruction. The key idea is to embed a single torn-bit within each atomically accessed data location in a RISE. The torn-bit's value is tracked per RISE and flipped at the beginning of each RISE. If all memory access instructions within a RISE share the same torn-bit value, the RISE is considered to have executed atomically. This mechanism is referred to as the *Torn-bit for Store Atomicity (TSA)*. For load instructions, we propose a mechanism called *Load-to-Persist (L2P)* technique, which immediately stores the loaded value into designated persistent storage, embedding the torn-bit accordingly. As elaborated in Section 3.5, atomicity checking for store instructions does not impact the correct recovery and thus is omitted for write-RISEs.

SSAPP utilizes a straightforward resumption algorithm, leveraging the type (read/write/RMW) and TSA status of the crashed RISE. If the TSA indicates the RISE was completed before failure, the recovery thread resumes execution from the "terminator" instruction (e.g., branch). SSAPP explicitly checks the RISE type, as re-executing a completed read-RISE or RMW-RISE may yield incorrect recovery. We discuss the details in Section 3.5.

3.1.2 Programmer's Responsibility.

Developers using SSAPP must: (1) specify the failure-atomic code region explicitly, (2) allocate persistent heap memory for memory accesses within those regions, and (3) ensure the software runs on persistent CPU cache hardware platforms such as Extended Asynchronous DRAM Refresh (eADR). SSAPP's effectiveness depends on the compiler's ability to recognize and persist transient states. For example, transient states available during runtime, such as Linux socket states, cannot be automatically persisted across crashes since they are not visible at compile-time. Persisting such states would require significant engineering effort or kernel-level redesign and remains future work. Furthermore, SSAPP assumes eADR-supporting hardware. Such hardware provisions enough power so the CPU can flush the data in a volatile cache before a complete blackout. However, assuming the input program is bug-free, SSAPP can safely handle crashes occurring even during recovery, as the same crash-consistency techniques used for pre-failure code are also present in the generated post-failure recovery code.

3.2 Partitioning into RISE

The post-failure program in the RVR model attempts to identify where the crash occurred and resume the execution until completion. However, tracking at every instruction incurs high overhead [19]. As with prior works [31, 50], SSAPP partitions the input program into an instruction sequence and tracks execution progress at a coarse granularity. Resuming at coarse granularity may result in redundant re-execution of instructions, and therefore, it is important to ensure that re-executions do not cause unintended side effects. Previous works referred to such re-executable code sequences as idempotent regions. SSAPP refines this concept of idempotency and introduces a complementary property called failure reentrancy.

To achieve piecewise failure reentrancy and idempotency, SSAPP splits a BB into one of the three types: read, write, or read-modify-write (RMW). When splitting a BB, SSAPP inserts unconditional branch instruction to preserve the original control and data flow. Additionally, SSAPP treats certain instructions such as `CallInst`, RMW, and memory barrier (e.g., `sfence`) as splitting boundaries. The rationale for splitting immediately after these instructions is that if these instructions are successfully executed, the whole RISE is successfully executed.

3.2.1 Identity, Failure Reentrancy, Idempotency.

First, we define the identity of two persistent images: two PMEM images are considered identical if the values at all corresponding offsets exactly match.

Failure Reentrancy An instruction sequence is failure-reentrant if it can resume from a partially persisted image — created due to a sudden crash — and produce a final memory image identical to that produced by uninterrupted execution. Formally, given a sequence of instructions $f(x)$ and an input x, let's assume that we partition $f(x)$ into $g(x)$ and $h(x)$, such that $f(x) = h(g(x))$. The sequence is failure-reentrant if, for all valid $h(x)$ and $g(x)$, the following property holds: $f(g(x)) = f(x)$.[1]

Idempotency An instruction sequence is idempotent if executing it multiple times results in an identical image. Formally, given a sequence of instructions $f(x)$ and an input image x, $f(x)$ is idempotent if $f(x) = f(f(x))$.

An alternative definition is an instruction sequence without an anti-idempotency pattern. An *anti-idempotency pattern* is Write-After-Read (WAR) data dependency [31]; if a program loads from memory and stores an updated value to the same memory location, it is not idempotent (e.g., `i++`). To eliminate these anti-idempotency patterns, SSAPP leverages that LLVM's BB is an instruction sequence that can easily be partitioned into two regions with an unconditional branch instruction without altering the control flow. After partitioning, each RISE is idempotent at a memory operation level since RISE exclusively reads or writes to the memory but never both. For an indivisible memory operation such as RMW, SSAPP employs explicit atomicity checks to avoid unintended re-execution (Section 3.4.1).

[1]Failure reentrancy can be defined for identity functions as long as the memory image is large enough to specify the intermediate status. For example, failure-reentrant identity function for a single-bit memory is impossible because there are not enough bits to represent an intermediate state.

3.3 Persisting Transient Data

Persisting transient state is essential for RVR. However, naive techniques like write-ahead logging or checkpointing introduce significant performance overhead by halting execution until data persistence completes. Instead, SSAPP efficiently persists transient data by redirecting volatile stack allocations to persistent memory. Specifically, SSAPP leverages LLVM's requirement that all virtual registers be explicitly defined using the `alloca` instruction on the stack. It then replaces these volatile stack allocations with persistent memory allocations, effectively redirecting transient stack data directly to persistent storage without incurring costly stalls or logging overhead.

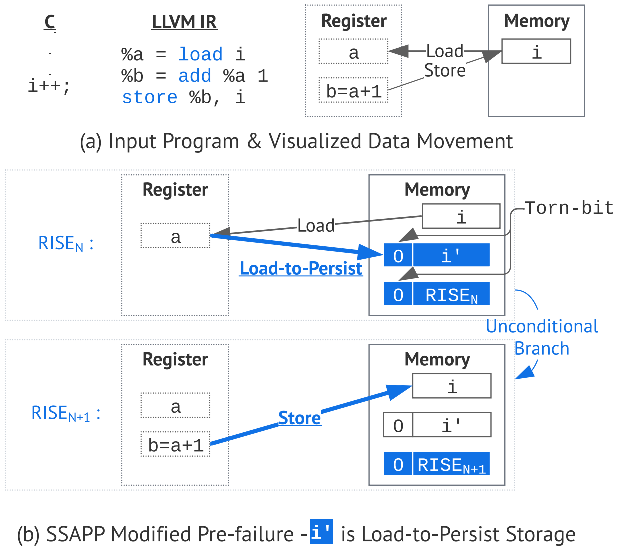
3.4 Embedding Metadata for Atomicity Check

Persisting transient data alone does not guarantee that a RISE can be correctly resumed. For instance, consider a scenario in which a write-RISE following a read-RISE overwrites the overlapping memory before a crash. Naively resuming from the read-RISE could lead to loading an updated, incorrect value, causing faulty recovery. The difficulty comes from accurately tracking program progress, since a crash could occur before progress-tracking information is updated, resulting in stale tracking. Incorrect tracking could misinform the recovery thread, resulting in incorrect resumption.

Instead of tracking the exact program progress, SSAPP tolerates approximate progress tracking via per cacheline single-bit versioning, known as the *torn-bit*. During recovery, this torn-bit hints at whether a set of memory accesses were executed atomically before a failure, termed the *Torn-bit for Store Atomicity (TSA)* check. For load instructions—which inherently do not store values—we propose a novel technique called *Load-to-Persist (L2P)*. L2P immediately persists each loaded value into a dedicated PMEM location, with a properly updated torn-bit. This approach simultaneously provides atomicity verification for load instructions and ensures the latest loaded values persist correctly. For approximate progress tracking, we propose a *Crash Buoy (CB)*, which holds both the RISE ID representing approximate progress and the corresponding ground-truth single-bit version information. We refer to the RISE ID indicated in CB as *CBID*.

3.4.1 Torn-Bit for Store Atomicity (TSA).

Checking the atomicity of a crashed RISE is challenging due to a potentially outdated CBID. As shown in Table 2, the CBID may be outdated by up to one RISE interval. For example, a pre-failure thread may complete execution of $RISE_N$ and entered the subsequent $RISE_{N+1}$. If a crash occurs before the CBID update, the recovery thread would perceive $RISE_N$ as the



(a) Input Program & Visualized Data Movement

(b) SSAPP Modified Pre-failure - i' is Load-to-Persist Storage

Figure 2. Load-to-Persist Example. Visualization of how SSAPP transforms a simple increment (i++) into two RISEs with explicit data persistence. Other SSAPP modifications are omitted for clarity. The transformation is similar to LLVM's register demotion, but instead redirects values into SSAPP-managed non-volatile memory with torn-bits applied for atomicity tracking.

crashed RISE. To mitigate the obscurity, we introduce the torn-bit mechanism.

A torn-bit is a special single-bit flag allocated for each SSAPP-managed memory-access destination. The post-failure recovery inspects the torn-bit of every SSAPP-managed memory location within a RISE to verify if RISE is atomically executed. If all locations bound to a RISE have the matching torn-bit as the CB's torn-bit, it confirms that the RISE executed atomically.

Optimization for Memcpy For dynamically allocated shadow memory or a large memory-copy operation (memcpy()), adding torn-bit operations for every 8-byte cache line incurs significant overhead. Instead, we allocate a shadow memory for the checksum of the memory region storing a checksum of the entire memory region along with a torn-bit. During recovery, a runtime library function compares the stored checksum and torn-bit against freshly computed values. A match indicates atomic execution, thus efficiently verifying RISE atomicity.

3.4.2 Load-to-Persist (L2P) for Load Atomicity.

The L2P technique persists any loaded value immediately into SSAPP-managed persistent storage, embedding an appropriate torn-bit. The post-failure program checks the atomicity for a set of L2P using the Torn-bit Store Atomicity (TSA, Section 3.4.1). If L2P operations are not atomically executed before a failure, the post-failure program can re-execute the L2P because the original load memory location remains unchanged (no overwriting update operations due to RISE decomposition; RMW is a special case as elaborated in Section 3.6). With L2P, SSAPP can track whether a set of load instructions was successfully executed before the failure by checking the embedded version information of corresponding PMEM memory.

Figure 2 (a) illustrates an example of L2P handling a simple increment (i++). A visual representation of i++ is shown on the right. A thread loads the value of i into register $\%a$, computes $\%b$ by adding 1, and stores $\%b$ back to the memory location i. This is a Write-After-Read (WAR) dependency because executing the store instruction overwrites the original memory location.

Figure 2 (b) visualizes how L2P resolves the WAR dependency and assists the atomicity check of a read RISE. In $RISE_N$, a thread reads from memory location i and persists to memory location i', hence the name Load-to-Persist. In the subsequent RISE, $RISE_{N+1}$, the thread writes the incremented value to the original destination i. The atomicity of read accesses in $RISE_N$ can be checked with TSAs, which are set to 0 in the figure. The added instructions do not impact the original program's control flow, data flow, or data layout (i' resides in SSAPP-managed non-volatile memory).

3.4.3 Tracking Program Progress with Crash Buoy.

SSAPP allocates per-thread 8-byte persistent storage for holding the RISE ID of the most recently executed RISE. Upon entering a new RISE, SSAPP stores its ID in this dedicated memory, referred to as the *Crash Buoy*. The value stored in CB is termed the *CBID*. SSAPP intentionally avoids enforcing strict persist ordering relative to other memory operations within a RISE, allowing for low overhead. A memory barrier placed at the end of each RISE sufficiently constrains potential reordering, ensuring consistent and correct tracking, as detailed in Table 2.

Table 2. Inferred Crash Point from CB and TSA.

Crash Buoy	TSA Status	Actual Crash		
		$RISE_{N-1}$	$RISE_N$	$RISE_{N+1}$
$RISE_N$	True	X	O	O
$RISE_N$	False	X	O	X

(O: possible X: impossible)

3.5 Recovery Algorithm

The modifications discussed earlier provide sufficient information to determine the correct resumption point after a crash. Here, we describe how SSAPP computes the *resumable RISE*, defined as the RISE from which resuming execution leads to an identical final memory state compared to uninterrupted execution. This computation depends on three key factors: the Crash Buoy (CB), the crashed RISE's type (read,

Table 3. Correct Resumption for Possible Cases Derived from Table 2.

RISE Type	CB	TSA	Actual Crash	Resume at N?	Resume at $N+1$?
Read	$RISE_N$	T	$RISE_N$	O	O
Read	$RISE_N$	T	$RISE_{N+1}$	X	O
Read	$RISE_N$	F	$RISE_N$	O	X
Write	$RISE_N$	T	$RISE_N$	O	O
Write	$RISE_N$	T	$RISE_{N+1}$	O	O
Write	$RISE_N$	F	$RISE_N$	O	X
RMW	$RISE_N$	T	$RISE_N$	Impossible	
RMW	$RISE_N$	T	$RISE_{N+1}$	X	O
RMW	$RISE_N$	F	$RISE_N$	O	X

(O: resumable, X: incorrect resumption)

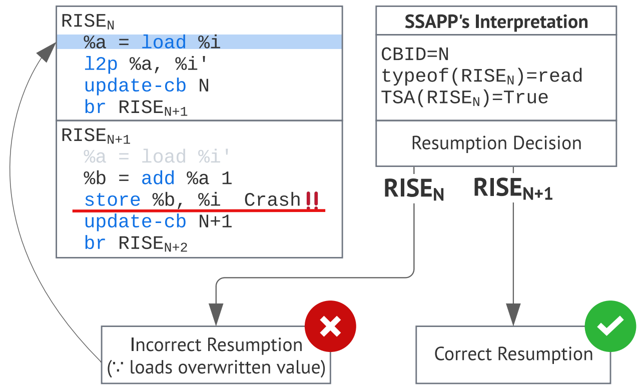


Figure 3. Example for Highlighted Cell in Table 3. Continuing with the `i++` example, the left pane shows the simplified IR-like code after SSAPP conversion. If a crash occurs after the `store` instruction in $RISE_{N+1}$, the SSAPP would interpret the PMEM state as shown under SSAPP's Interpretation. Naively resuming from $RISE_N$ is incorrect because `load` instruction reads an already overwritten value, violating failure reentrancy. Instead, correctly resuming from $RISE_{N+1}$ avoids duplicate execution and preserves program semantics.

write, or RMW), and the Torn-bit Store Atomicity (TSA) status.

Why Naive Resumption Fails. Even under ideal conditions — where all of the RISEs are piecewise failure-reentrant and idempotent, and CB updates in order relative to other store instructions — naively resuming from the crashed RISE can lead to incorrect recovery. Without loss of generality, we assume N for CBID and $N-1$ and $N+1$ as RISE ID of predecessor and successor RISEs, respectively. Figure 2 shows that CB can be outdated if updating the CB to $N+1$ is reordered with $RISE_{N+1}$ execution. An outdated CBID becomes problematic when $RISE_N$ and $RISE_{N+1}$ collectively form an anti-idempotency pattern. For example, if $RISE_N$ reads a transient value and $RISE_{N+1}$ increments and writes the updated value, resuming from $RISE_N$ would incorrectly increment the value twice.

Correct Resumable RISE Computation. To accurately identify the resumable RISE, the recovery logic inspects the CB, determines the crashed RISE's type, and evaluates the TSA status. As shown in Table 3, re-executing the RISE indicated by CBID is incorrect when it is write-RISE and has completed atomic checkpointing (i.e., TSA = `True`). Based on this insight, SSAPP uses the following simple algorithm.

- If the crashed RISE (CBID) is of type read or RMW and the TSA check passes (TSA = true), the recovery logic resumes execution at the end of this RISE.
- In all other scenarios, the recovery logic resumes from the beginning of the crashed RISE.

3.6 Correctness

We validate SSAPP's correctness along three dimensions: (1) maintaining the original input program's control flow, data flow, and data layout, (2) preserving TSA and L2P data in the presence of power failure, and (3) resuming execution at a correct RISE. More rigorous proof is available in Appendix A.

Preserved Control Flow/Data Flow/Data Layout. We preserve the control flow and data flow of the original program by inserting non-intrusive instructions: (1) unconditional branch to split the basic block, (2) store instructions to newly allocated destination, and (3) no modification to original store instructions. While the data layout may change slightly due to new allocations, these modifications do not impact the original program logic. Such changes only provide additional hints to the SSAPP-aware post-failure thread regarding the atomicity of loaded values.

Persistence Across Power Failure To preserve the control flow and data flow across power failures, SSAPP ensures that volatile transient data is persisted. SSAPP assumes that the heap is persistent and stack variables are all loaded before their use (following LLVM's Single Static Assignment rules). Therefore, SSAPP allocates and persists every loaded value, effectively preserving the transient state across sudden crashes.

Recovery Algorithm Correctness We demonstrate the correctness of resumable RISE computation using the tables (Table 2, Table 3) and an example illustration (Figure 3) of an incorrect resumption for a naive approach. If the two subsequent RISEs do not access the same memory location, resuming execution directly from the CBID is correct. We assume in Table 2 that $RISE_N$ and $RISE_{N+1}$ access a shared memory location.

As shown in the Table 2, a fence at the end of each RISE limits the possible states inferred from the CB and the TSA status. At the moment of a power failure, suppose the thread was executing $RISE_N$. The TSA status could be either of two states: `True` indicating that all of the memory accesses within the $RISE_N$ were atomic, or `False` indicating that at least one of them was incomplete. In either case, the actual failure point (as opposed to the CB, which is only an approximate failure point) can not be $N-1$ because overwriting the CB with value N must occur after all preceding stores. Additionally, out-of-order execution across RISE is infeasible due to the last fence instruction at each RISE's end. If the TSA status is `False`, the actual failure point can not be $N+1$ either.

Table 3 enumerates all possible combinations of RISE types, CB values, TSA statuses, and the actual failure point, marking the correct resumption as O and incorrect resumptions as X. Because each RISE is idempotent and failure-reentrant and the CB suggests that the failure point is at least N, resuming from N is typically correct. Exceptions occur when RISE is a read-RISE or RMW-RISE and TSA status is `True`. In such cases, if the actual failure point was $N+1$ with most of the store instructions completed except for CB overwrite, the shadow memory referenced in $RISE_N$ could have been updated already. In such a scenario, resuming from the $RISE_N$ would result in loading the already updated value. This leads to re-executing the $RISE_{N+1}$, violating correctness. Furthermore, if the TSA status is `False`, the recovery program should not resume from $RISE_{N+1}$, since the $RISE_N$ was incomplete at failure time.

For RMW-RISE, resumption at $RISE_N$ with TSA `True` is incorrect because SSAPP guarantees the RMW instruction is always the final instruction of an RMW-RISE. Prior work has shown that RMW instructions internally execute serializing instructions, ensuring all stores become globally visible [46]. Thus, resuming at ($RISE_N$, RMW, `True`) would incorrectly execute the RMW operation twice. Conversely, the state ($RISE_N$, RMW, `False`) must not resume at $RISE_{N+1}$, as doing so would skip the necessary RMW operation, resulting in incorrect behavior.

In Figure 3, we clarify with a concrete example where naive resumption from the CB is incorrect. The example code is a simple increment implementing the write-after-read (WAR) pattern. SSAPP would split it into two RISEs and let us assume that a failure occurred after finishing most of the $RISE_{N+1}$ execution but before updating the CB. The post-failure program reads CB as N with TSA status `True`. However, resuming from $RISE_N$ is incorrect because it would load the already updated value from `%ptr` and increment it again in $RISE_{N+1}$, leading to erroneous results. Hence, in this scenario, resuming at $RISE_N$ is incorrect.

4 Implementation

This section details the implementation of RISE creation, data persistence (transient data, TSA, L2P, CB), and resumable RISE computation from TSA and CB. The **pre-failure converter** transforms the input into a sequence of RISEs and applies SSAPP techniques. The **post-failure generator** inserts the resumption computation logic on top of the modified pre-failure code. These compiler extensions output pre-failure and post-failure LLVM IR, respectively. The generated IR invokes SSAPP library functions. The **SSAPP runtime library** manages function invocations, thread context, and allocated shadow memory.

4.1 Pre-Failure Code Converter

We elaborate on the pre-failure code modifications, including (1) creating a conversion-ready LLVM IR from the source code, (2) converting each basic block (BB) to RISE, (3) redirecting transient data to PMEM-backed memory (including Load-to-Persist), (4) persisting program progress on CB, and (5) embedding TSA. The SSAPP converter retains the input program's control flow and data flow by merely inserting non-destructive instructions such as the unconditional branch. When memory writes are necessary, they target newly allocated memory, preserving the original program's data layout.

Inlining Call Sites As the pre-failure modifier iterates over each BB within a failure atomic function, a called function may not be modified. However, all function call sites within the programmer-delineated region must also be idempotent from the memory access perspective. For functions with source code available, SSAPP inlines them as part of the FASE and processes them as if they were part of the parent function. For external functions without available source code, SSAPP requires reentrancy and idempotency at the PMEM level.

Register Demotion The SSAPP converter may miss transient data passed across RISEs via registers. Fortunately, LLVM provides a transformation pass called `-reg2mem`, which restricts inter-BB data flow to the stack. In LLVM IR, stack values are allocated using the `alloca` instruction. Unlike virtual registers, which are transparently mapped to the physical register during object file compilation, this stack memory is explicitly allocated, stored, and loaded. With explicitly named variables that span across RISEs, SSAPP can naturally identify the "live-in" and "live-out" data for each RISE and pre-allocate the shadow memory for each data.

Converting Basic Block to RISE To provide an underlying assumption for correct resumable RISE computation, each RISE either reads or writes data to memory but not both, with RMW as an exception. The key idea is eliminating the Write-After-Read (WAR) pattern by separating a sequence of load instructions with an unconditional branch. SSAPP uses RISE delimiters such as `sfence`, function calls, atomic operations, and the tail instruction of load sequences. For each delimiter, SSAPP adds an unconditional branch to split them into two back-to-back basic blocks.

TSA Implementation All types smaller than 8 bytes are cast to an 8-byte type and use the Most Significant Bit (MSB) as the torn-bit. An 8-byte type is split into two 4-byte integers where each is stored in 8-byte storage with the MSB being the torn-bit. A RISE is considered atomically executed if all torn-bits match the CB's torn-bit.

SSAPP maintains a per RISE *torn-bit flipper*, an 8-byte value initialized with all bits set to 1 except the MSB. The MSB is flipped within each RISE, and a bitwise `xor` is applied to the data so that all stores within the RISE have the same torn-bit (MSB of data is always zero, due to zero-extension, `zext`, or logical shift right, `lshr`). Furthermore, flipping per entry distinguishes one iteration from another even when a single RISE is executed back-to-back.

In theory, SSAPP sets a torn-bit for every 8-byte memory access. However, in practice, it is unnecessary to apply TSA to all memory accesses. As shown in the middle three rows of Table 2, a recovery thread can naively resume from the write RISE regardless of the TSA status. Thus, we only apply TSA to read RISE and RMW RISE. For RMW operations, SSAPP checks the operand bit-width to determine if TSA embedding is possible.

TSA for RMW Fallback Plan SSAPP may attempt to use an RMW with a wider bit-width than the original and repurpose one of the unused bits as a torn-bit. This may alter the original program's data layout and must therefore be applied cautiously. If RMW with a wider bit-width is not supported, SSAPP aborts. Implementing such a fallback plan is left for future work.

Load-to-Persist Insertion At its core, SSAPP persists each loaded value to the corresponding shadow stack. For each load instruction in read RISE, SSAPP inserts store instructions that take the loaded value and the pre-allocated L2P storage as operands. We apply TSA to L2P as well.

Crash Buoy Insertion Updating the Crash Buoy is as simple as `or`'ing the RISE ID with the torn-bit flipper and storing it in the per-thread Crash Buoy storage. The RISE ID is hard-coded.

Adding Trailing Fence SSAPP adds fence instructions for each RISE if it does not already have a trailing memory barrier. With the trailing fence, all writes are globally visible before executing the subsequent RISE.

4.2 Post-failure Generator

The post-failure generator inserts instructions to read the per-thread contexts, determines the TSA state, computes the resumable RISE, and resumes execution. To handle ambiguous RISE boundaries (e.g., `ConditionalBr`), the post-failure generator may split a RISE to resolve the ambiguity.

Loading Per-Thread Context The generated code inserts operations to load and inspect the thread context (FASE ID, RISE ID, variable-to-storage mapping, CB, etc.). This process includes restoring the shadow stack, CBID, and RISE flippers.

Computing Resumable RISE If the crashed RISE is not a read RISE, execution always resumes from the current CBID without checking the TSA. If the crashed RISE is a read RISE, SSAPP invokes a runtime library function to decide whether to resume execution from the crashed RISE or skip it based on the TSA inspection outcome.

Resuming Execution The generator inserts a conditional branch based on the resumption decision, which leads to either one of the two `switch` statement versions: one that jumps to the crashed RISE or one that jumps to the successor RISE. Each switch statement compares the CBID against all possible RISEs and jumps to their corresponding hard-coded destinations.

4.3 Runtime Library

The runtime library contains reusable logic across workloads: thread context management, shadow memory management, and the resumption computation algorithm.

Thread Context Management Each thread maintains its context, including the valid bit, thread ID, currently executing FASE ID, Crash Buoy, and shadow memory mapping information.

Shadow Memory Management Each type of shadow memory is associated with two files: mapping data and actual data. Mapping data maps the shadow ID, which is a tuple consisting of (FASE ID, RISE ID, Inst ID), to the offset of allocated memory within the data file.

Resumption Computation The algorithm's key decision is whether to re-execute the crashed RISE, based on the TSA state. To check the TSA state, the runtime library iterates the shadow memory associated with the CBID. If the torn-bits of all associated shadow memory match the CB torn-bit, the function returns `false`, indicating that execution should resume from the successor of the crashed RISE.

5 Evaluation

To evaluate SSAPP's performance improvement, we use a variety of benchmarks including basic data structures (B+-tree, RB-tree, hashmap, skiplist), transactional benchmark (TATP, TPC-C, Vacation), and lock-free data structures (list, hash, binary-search tree).

5.1 Configuration

We evaluate our work on a single-socket system with an Intel Xeon Gold 6230 CPU, which has 20 physical cores each with a 32 KiB L1 cache. We used a 2:2:2 topology where each DRAM/PMEM channel has 16 GB, and 128 GB capacity, respectively (a total of 96 GB/768 GB of DRAM/PMEM). All binaries were compiled using clang version with `-O1` optimization flag. We emulated the eADR environment by eliminating cacheline flushes at the program language level.

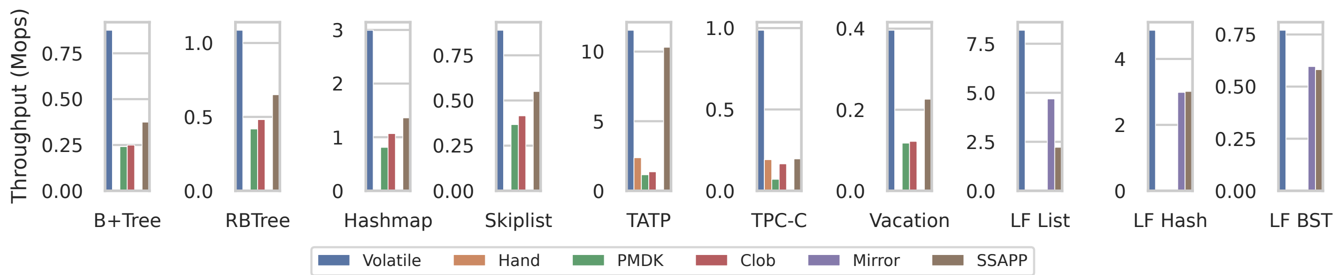


Figure 4. Throughput of Evaluated Workloads. This figure compares the single-threaded throughput (in millions of operations per second, MOPS) across various workloads and crash consistency mechanisms. Empty bars indicate inapplicable configurations, such as applying Clobber-NVM (Clob) to lock-free (LF) data structures. `Volatile` refers to the baseline version of the program, running on volatile memory without any crash consistency mechanisms. Hand represents a hand-optimized implementation in which the programmer pre-allocates dedicated logging memory per thread and per transaction type; this is applicable only to the TATP and TPC-C benchmarks.

5.2 Performance Overhead

To assess SSAPP's performance, we evaluated four data structures, three transactional benchmarks (TATP, TPC-C, Vacation), and three lock-free data structures (list, hash, binary-search tree). Missing bars indicate inapplicable cases, such as applying Clobber-NVM [50] to lock-free data structure or applying Mirror [7] to transactional benchmarks. We used the same configuration, parameters, and workloads as in prior works [7, 50]: 50:50 read/write ratio for B+Tree, RB-Tree, Hashmap, Skiplist, subscriber transaction for TATP, new order transaction for TPC-C, make reservation transaction for Vacation, and 80:20 read/write ratio for lock-free data structures. Figure 4 shows the single-threaded raw throughput of workloads in million operations per second (MOPS). We make the following observations.

First, for basic data structures (B+Tree, RBTree, Hashmap, Skiplist), SSAPP outperforms Clobber-NVM by 1.4×. The reduced number of memory barriers and higher cache locality contribute to the performance advantage of SSAPP.

Second, SSAPP outperforms Clobber-NVM for TATP and TPCC by 7.5× and 1.2×, respectively. SSAPP shows significantly greater performance improvement for TATP than for TPC-C. This is because TATP transactions are smaller than those in TPC-C, resulting in fewer data loads. With fewer load instructions, SSAPP performs fewer basic block (BB) splits. Since SSAPP's overhead is proportional to the number of BBs (e.g., storing Crash Buoy) its overhead for TATP is smaller than in prior work.

Third, as shown in TATP and TPC-C results, programmer-tuned persistence (Hand) outperforms the PMEM programming library and Clobber-NVM. However, SSAPP achieves similar performance to hand-tuned solutions without the programming burden.

Fourth, for the vacation benchmark, a workload with frequent queries, SSAPP achieved 1.9× higher throughput than the Clobber-NVM. The frequent query within the transaction resulted in frequent clobber writes where Clobber-NVM would append a log entry in PMEM. PMDK showed a nearly identical performance to Clobber-NVM because PMDK had only a few additional logging to make when compared to the Clobber-NVM. Unlike these logging-based approaches, SSAPP does not create a log entry for persistence. For SSAPP, a clobber write is translated into an additional unconditional branch instruction and store operations to shadow memory. As these shadow memories have high temporal locality, SSAPP outperformed the logging-based approaches for the vacation benchmark.

Lastly, SSAPP performs nearly as fast as a hand-crafted persistent lock-free data structure [7] that does not fit on the L1 cache. If the data fits on the L1 cache as in Mirror-List, SSAPP performs worse because additional writes (e.g., Load-to-Persist) impose higher pressure on the L1 data cache, frequently causing L1 cache misses. We measured the average number of instructions executed per operation (insert, remove, read). We found that the SSAPP version has about 4.8× more instructions than Mirror but executes at 2.3× higher IPC, resulting in a total of 2× slowdown.

Based on these observations, we conclude that SSAPP can transparently provide crash consistency for both the transactional and non-transactional workloads with minimal performance overhead.

5.3 Scalability

To understand the scalability of SSAPP, we measured the throughput of four lock-based and lock-free data structures when exponentially increasing the number of threads from 1 to 16. The transactional benchmarks were excluded from the scalability evaluation because the available implementations did not support concurrency. We report raw throughput in a million operations per second. We make the following observations from the results.

For four lock-based data structures, SSAPP showed 1.4× higher throughput than Clobber-NVM when varying the

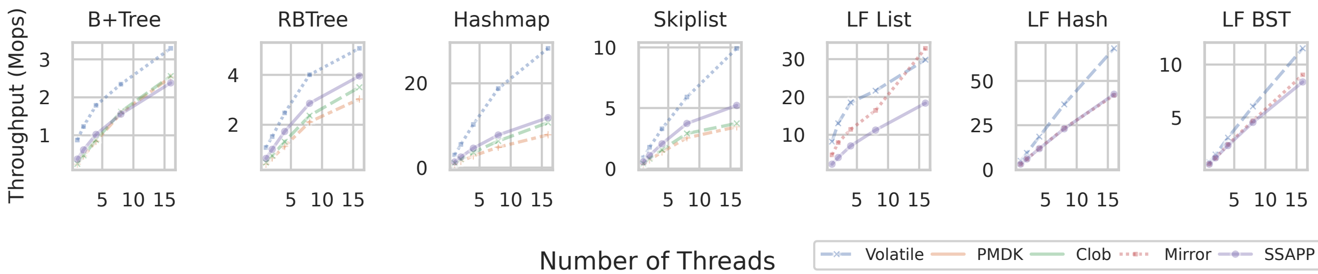


Figure 5. Scalability of Workloads. Throughput scalability (in MOPS) of various workloads is shown as the number of threads increases from 1 to 16. The left four panels represent lock-based data structures (B+Tree, RBTree, Hashmap, Skiplist), while the right four represent lock-free (LF) data structures (List, Hash, BST). `Volatile` represents the baseline program running on DRAM without persistence mechanisms. `Clob`, `PMDK`, and `Mirror` are prior crash-consistency approaches, and `SSAPP` is the proposed system. SSAPP achieves comparable or better scalability than prior solutions while maintaining crash consistency guarantees.

number of threads. When compared against the volatile version, a vanilla program executed on DRAM, SSAPP closed the gap to 60% across data structures, up from 52% for Clobber-NVM.

For lock-free data structures, SSAPP showed comparable scalability to Mirror for `Hash` and `BST`. SSAPP exhibited a 2% overall slowdown for these two workloads. As discussed in Section 5.2, `List` performs well for Mirror, even better than the volatile version, as observed in the original paper. Regardless of the absolute performance, SSAPP scales well even for such a short function with RMW instructions.

5.4 Litmus Example Validation

We designed a framework to validate crash consistency using small litmus tests. We implemented a transformation pass for crash consistency validation purposes. It makes the following modifications to the pre-failure code: (1) inserts `printf` call instructions at the beginning of each RISE, which prints the FASE ID and RISE ID, (2) reads an integer environment variable called `SSAPP_STOPPER`, (3) inserts a conditional branch at the end of each RISE that exits the program if the `SSAPP_STOPPER` matches the RISE ID. Using the modified pre-failure code, the tester checks whether the post-failure thread can recover from the intentionally created inconsistent program state. A successful recovery yields a program state identical to that produced by the vanilla program. For all possible RISEs in each litmus program, SSAPP passed the crash consistency test.

6 Related Work

6.1 Register-Level Persistence Support

Various works assume or propose register-checkpointing to prevent stale load after a power failure [21, 53]. However, this involves splitting the program into fine-grained regions, typically limited by the number of physical registers. This limitation motivated static analysis of the input program to identify regions where the compiler pass inserts a `fence` instruction.

ReplayCache leveraged the recovery-via-resumption to enable volatile caches for non-volatile memory [53]. It uses Just-In-Time register checkpointing to preserve the store operands and re-execute store instructions during recovery. When ReplayCache detects a potential register stack spill, it inserts a `fence` instruction to ensure that store operand data remains in registers. This approach is suitable for systems without volatile memory support but with Just-In-Time register snapshots. However, frequent `sfence` instructions can degrade performance. Capri [21] also provides register-level replayability, but requires undo+redo logging for persistence and architectural modifications to support register checkpointing.

6.2 Testing, Debugging, and Automation Tools

PMEM programming is difficult; issuing flush and fence instructions carefully to ensure crash consistency without incurring high overhead is non-trivial. Overuse of flush and fence instructions is often referred to as a performance bug. Failure to ensure crash consistency under specific failure scenarios constitutes a correctness bug.

Many prior works have focused on testing and debugging PMEM programs. Researchers have explored various approaches, including brute-force/model checking [11, 18, 26], dynamic analysis [6, 34, 35], fuzzing [32] and symbolic execution [43]. Despite extensive research, new categories of correctness bugs continue to be discovered [3, 10, 12, 13, 27].

Recent work strives to automate PMEM programming to some extent, but shows clear limitations in terms of accuracy and performance overhead. Ayudante automatically inserts calls to the PMDK library — an industry-standard PMEM programming library — using a reinforcement-learning-based approach. However, it remains experimental due to limited training/testing data and lacks high accuracy [16]. Hippocrates attempts to automatically fix bugs by inserting flush and fence instructions. However, the programmer is still responsible for optimizing performance by removing redundant operations and writing post-failure recovery code [42].

6.3 Other Solutions for Persistence

Several works have proposed system-level solutions for persistence [14, 23, 24, 28, 30, 45, 49, 54, 58]. These systems aim to make persistence overhead transparent to the programmer by handling it at the system stack level. Many of these works target specific hardware configurations [49, 54, 58] and often proposed new interfaces [23, 24, 30].

Other works propose software-hardware co-designs that aim to simplify programming by offering a well-defined persistence model at low performance cost [1, 9, 20, 25, 33, 40, 52, 55, 57]. These approaches typically use a dedicated buffer to collect write operations before updating designated memory locations on PM. In addition, they extend the existing ISA with dedicated instructions to control persist ordering. However, many of these works require non-trivial hardware modifications such as an extended cache coherence protocol.

7 Conclusion

To benefit from a byte-addressable, non-volatile memory, a programmer-friendly, low-overhead crash consistency solution is needed. To address these limitations, we built SSAPP, a compiler extension that transparently adds crash consistency to a wide range of programs. SSAPP modifies the input program into a collection of failure reentrant, idempotent sections without affecting the input program logic or incurring high overhead. The generated recovery code recovers partially executed program states into a crash consistent state by diligently resuming the execution from where a sudden power failure may have occurred. SSAPP bridges the gap between performance and reliability, providing a practical path to robust persistent memory programming.

Acknowledgments

This paper is supported by the PRISM and ACE centers in JUMP 2.0, an SRC program sponsored by DARPA.

References

[1] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 111–124. https://doi.org/10.1109/HPCA51647.2021.00019

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

[3] 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 '22)*. Association for Computing Machinery, New York, NY, USA, 873–887. https://doi.org/10.1145/3503222.3507755

[4] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 105–118. https://doi.org/10.1145/1961295.1950380

[5] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. 2020. Lazy Release Persistency. 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, 1173–1186. https://doi.org/10.1145/3373376.3378481

[6] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. *Fast, Flexible, and Comprehensive Bug Detection for Persistent Memory Programs*. Association for Computing Machinery, New York, NY, USA, 503–516. https://doi.org/10.1145/3445814.3446744

[7] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: Making Lock-Free Data Structures Persistent. 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, 1218–1232. https://doi.org/10.1145/3453483.3454105

[8] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-Free Regions. *SIGPLAN Not.* 53, 4 (jun 2018), 46–61. https://doi.org/10.1145/3296979.3192367

[9] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2020. Relaxed Persist Ordering Using Strand Persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 652–665. https://doi.org/10.1109/ISCA45697.2020.00060

[10] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. 2022. 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

[11] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2021. Jaaru: Efficiently Model Checking Persistent Memory Programs. 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, 415–428. https://doi.org/10.1145/3445814.3446735

[12] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2022. 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

[13] Zhilei Han and Fei He. 2025. Robustness Verification for Checking Crash Consistency of Non-volatile Memory. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 955–969. https://doi.org/10.1145/3669940.3707269

[14] George Hodgkins, Yi Xu, Steven Swanson, and Joseph Izraelevitz. 2023. Zhuque: Failure is Not an Option, it's an Exception. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 833–849. https://www.usenix.org/conference/atc23/presentation/hodgkins

[15] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-Threaded Applications. In *Proceedings of the Twelfth European*

Conference on Computer Systems (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 468–482. https://doi.org/10.1145/3064176.3064204

[16] Hanxian Huang, Zixuan Wang, Juno Kim, Steven Swanson, and Jishen Zhao. 2021. Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 789–804. https://www.usenix.org/conference/atc21/presentation/huang-hanxian

[17] Intel Coporation. 2019. PMDK. https://pmem.io/pmdk/

[18] Intel Corporation. 2019. PMReorder. https://pmem.io/pmdk/pmreorder

[19] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 427–442. https://doi.org/10.1145/2980024.2872410

[20] Jungi Jeong and Changhee Jung. 2021. PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 517–529. https://doi.org/10.1145/3445814.3446698

[21] Jungi Jeong, Jianping Zeng, and Changhee Jung. 2022. Capri: Compiler and Architecture Support for Whole-System Persistence. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (Minneapolis, MN, USA) *(HPDC '22)*. Association for Computing Machinery, New York, NY, USA, 71–83. https://doi.org/10.1145/3502181.3531474

[22] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 361–372. https://doi.org/10.1109/HPCA.2017.50

[23] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 494–508. https://doi.org/10.1145/3341301.3359631

[24] Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. 2020. SubZero: Zero-Copy IO for Persistent Main Memory File Systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (Tsukuba, Japan) *(APSys '20)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3409963.3410489

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

[26] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 433–438. https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

[27] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. 2023. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 718–733. https://doi.org/10.1145/3552326.3567498

[28] Hayley LeBlanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. 2024. SquirrelFS: using the Rust compiler to check file-system crash consistency. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 387–404. https://www.usenix.org/conference/osdi24/presentation/leblanc

[29] 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

[30] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. 2022. ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 35–50. https://www.usenix.org/conference/fast22/presentation/li

[31] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 258–270. https://doi.org/10.1109/MICRO.2018.00029

[32] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. 2021. *PMFuzz: Test Case Generation for Persistent Memory Programs*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3445814.3446691

[33] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. 2019. Janus: Optimizing Memory and Storage Support for Non-Volatile Memory Systems. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 143–156.

[34] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. 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, 1187–1202. https://doi.org/10.1145/3373376.3378452

[35] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 411–425. https://doi.org/10.1145/3297858.3304015

[36] Suyash Mahar, Sihang Liu, Korakit Seemakhupt, Vinson Young, and Samira Khan. 2021. Write Prediction for Persistent Memory Systems. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 242–257. https://doi.org/10.1109/PACT52795.2021.00025

[37] Suyash Mahar, Mingyao Shen, Terence Kelly, and Steven Swanson. 2023. Snapshot: Fast, Userspace Crash Consistency for CXL and PM Using msync. arXiv:2310.16300 [cs.DC] https://arxiv.org/abs/2310.16300

[38] Suyash Mahar, Mingyao Shen, TJ Smith, Joseph Izraelevitz, and Steven Swanson. 2024. Puddles: Application-Independent Recovery and Location-Independent Data for Persistent Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 575–589. https://doi.org/10.1145/3627703.3629555

[39] 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

[40] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[41] 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

[42] Ian Neal, Andrew Quinn, and Baris Kasikci. 2021. Hippocrates: Healing Persistent Memory Bugs without Doing Any Harm. 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, 401–414. https://doi.org/10.1145/3445814.3446694

[43] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1047–1064. https://www.usenix.org/conference/osdi20/presentation/neal

[44] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 336–349. https://doi.org/10.1109/HPCA.2018.00037

[45] Yanqi Pan, Hao Huang, Yifeng Zhang, Wen Xia, Xiangyu Zou, and Cai Deng. 2024. Delaying Crash Consistency for Building A High-Performance Persistent Memory File System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 9 (2024), 2620–2634. https://doi.org/10.1109/TCAD.2024.3375792

[46] Yun Joon Soh, Steven Swanson, and Jishen Zhao. 2024. ENTS: Flush-and-Fence-Free Failure Atomic Transactions. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, VA, USA) *(MEMSYS '23)*. Association for Computing Machinery, New York, NY, USA, Article 25, 16 pages. https://doi.org/10.1145/3631882.3631907

[47] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. *SIGARCH Comput. Archit. News* 39, 1 (mar 2011), 91–104. https://doi.org/10.1145/1961295.1950379

[48] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye. 2020. MorLog: Morphable Hardware Logging for Atomic Persistence in Non-Volatile Main Memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 610–623. https://doi.org/10.1109/ISCA45697.2020.00057

[49] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 478–496. https://doi.org/10.1145/3132747.3132761

[50] 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. https://doi.org/10.1145/3445814.3446730

[51] Yi Xu, Suyash Mahar, Ziheng Liu, Mingyao Shen, and Steven Swanson. 2024. CXL Shared Memory Programming: Barely Distributed and Almost Persistent. arXiv:2405.19626 [cs.DC] https://arxiv.org/abs/2405.19626

[52] Sujay Yadalam, Nisarg Shah, Xiangyao Yu, and Michael Swift. 2022. ASAP: A Speculative Approach to Persistence. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 892–907. https://doi.org/10.1109/HPCA53966.2022.00070

[53] Jianping Zeng, Jongouk Choi, Xinwei Fu, Ajay Paddayuru Shreepathi, Dongyoon Lee, Changwoo Min, and Changhee Jung. 2021. ReplayCache: Enabling Volatile Cachesfor Energy Harvesting Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 170–182. https://doi.org/10.1145/3466752.3480102

[54] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 897–911.

[55] Ming Zhang and Yu Hua. 2023. Silo: Speculative Hardware Logging for Atomic Durability in Persistent Memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 651–663. https://doi.org/10.1109/HPCA56546.2023.10071034

[56] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 658–674. https://doi.org/10.1145/3600006.3613135

[57] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the performance gap between systems with and without persistence support. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 421–432.

[58] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 207–219. https://www.usenix.org/conference/fast19/presentation/zheng

Received 2025-03-18; accepted 2025-05-03