# Persistent State Machines for Recoverable In-memory Storage Systems with NVRam

Wen Zhang, *UC Berkeley;* Scott Shenker, *UC Berkeley/ICSI;*
Irene Zhang, *Microsoft Research/University of Washington*

https://www.usenix.org/conference/osdi20/presentation/zhang-wen

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

# Persistent State Machines for Recoverable In-memory Storage Systems with NVRam

Wen Zhang
*UC Berkeley*

Scott Shenker
*UC Berkeley/ICSI*

Irene Zhang
*Microsoft Research/University of Washington*

## Abstract

Distributed in-memory storage systems are crucial for meeting the low latency requirements of modern datacenter services. However, they lose all state on failure, so recovery is expensive and data loss is always a risk. Persistent memory (PM) offers the possibility of building fast, persistent in-memory storage; however, existing PM systems are built from scratch or require heavy modification of existing systems. To rectify these problems, this paper presents Persimmon, a PM-based system that converts existing distributed in-memory storage systems into persistent, crash-consistent versions with low overhead and minimal code changes.

## 1 Introduction

In the past decade, distributed in-memory storage systems have become ubiquitous. Facebook and Twitter have petabytes of in-memory storage [2, 75], and in-memory replicated systems such as NOPaxos [58] and TAPIR [113] can process transactions within microseconds while providing consistency and fault-tolerance. As datacenter networks become faster and kernel bypass removes OS bottlenecks, only in-memory storage systems will be able to keep up with network speeds.

Unfortunately, in-memory storage systems have a crucial drawback: their lack of durability means that failed nodes must recover from a replica or another source (e.g., a persistent back-end database), which can be extremely slow. For example, a Facebook memcached cluster can take hours to regain full capacity if repopulated from another "warm" cluster, or days if repopulated from backend storage [75]. Even worse, state can be permanently lost if all replicas crash such as in a full datacenter failure. To reduce the impact of failures, many popular in-memory systems (e.g., Redis [85], RAMCloud [76]) support persistence, but it requires additional, complex code and/or incurs high performance overhead.

Persistent memory (PM) offers a promising solution for in-memory services. It is durable, offers performance close to DRAM, and is increasingly available in large sizes. However, PM systems require crash consistency [7, 57, 71, 79] (i.e., no system invariants are violated on a crash), which is complicated and expensive to enforce. Maintaining crash consistency requires that operations are failure-atomic [45]; for example, on crashes, an operation's deallocations and pointer updates must either atomically succeed or fail to avoid violating the invariant that pointers do not point to deallocated memory.

To ensure failure atomicity, PM systems must carefully flush volatile CPU state at specific times and possibly use write-ahead logging or other techniques to correctly recover from failures. These added flushes and writes impose significant overhead. As a result, most existing PM storage systems are carefully written from scratch for correctness and performance; even then, none can achieve the performance of today's in-memory systems. Recent work, like RECIPE [57] and MOD [39], aim to reduce application complexity by converting existing data structures to persistence on PM; however, because they exploit certain data structure properties (e.g., non-blocking synchronization), they are not suited to all in-memory storage systems.

This paper aims to let existing in-memory storage systems more easily reap the benefits of persistent memory. We make the key observation that distributed systems are typically designed as RPC-processing state machines. State machines are an ideal abstraction for PM because: (1) they encapsulate application state for recovery; (2) their operations offer clear failure-atomic regions; and (3) their state can be recreated at any time by re-executing operations.

Based on this insight, we present Persimmon, a PM-based system that converts existing in-memory distributed storage systems into durable, crash-consistent versions with low overhead and minimal code changes. Persimmon offers a new abstraction for building PM applications: *persistent state machines* (PSM). PSMs offer a simple guarantee: once an operation on the PSM returns, its side-effects on the PSM will never be lost. PSM operations are also failure-atomic: if the operation did not return before a crash, either the entire operation will be applied after the crash or none of it. PSMs can run arbitrary application code; however, like other state machines (e.g., replicated state machines), PSM operations must not have external dependencies (e.g., they cannot open file descriptors), must be deterministic, and are executed sequentially. As a result, Persimmon does not support multi-threaded applications that apply operations concurrently.

To minimize the performance overhead of accessing PM on the request processing path, Persimmon keeps two state machine copies, one in DRAM and one in PM. When the application invokes a PSM operation, Persimmon first executes the operation on the DRAM copy. If the operation is read-only, Persimmon returns. If the operation is read-write, Persimmon persistently logs the operation before returning. This design limits the critical path to DRAM for read-only operations and one sequential write to PM for read-write operations; however, it requires both a DRAM and a PM state machine copy, which can be large for in-memory storage systems.

On failure, Persimmon can recover the PSM by replaying the persistent log. However, to minimize recovery time, Persimmon asynchronously keeps the persistent state machine snapshot in PM up-to-date. The state machine abstraction lets Persimmon update the PM snapshot with a *crash-consistent shadow execution* of each PSM operation, which is then removed from the log. This design is crucial for large in-memory storage systems that might have terabytes of data. To recover, Persimmon simply copies the PM snapshot to DRAM, processes the remaining persistent log, and restarts the application. Our design uses a background process, which runs on another CPU, to perform the shadow execution, trading off the use of a CPU for faster recovery times.

From this description, it is clear that Persimmon minimizes the overhead of persistence on the request processing path. However, to achieve reasonable recovery times, the crash-consistent shadow execution of the log must also be efficient, so the log does not grow too large. Most of the difficult technical challenges lie in optimizing this shadow execution, and we are not aware of similar work that addresses this particular issue. Note that despite these technical challenges, using a persistent log is preferable to checkpointing for large in-memory storage systems that might have terabytes of data.

We use Persimmon to persist two in-memory distributed systems: Redis and TAPIR [113]. We implement both systems on Linux and with kernel-bypass networking. We evaluate Persimmon on three servers with 3 TB of Intel® Optane™ DC Persistent Memory and found:

- On a 90% read-heavy YCSB workload, Persimmon incurs no discernible overhead to the latency and throughput of standard Redis; and near-zero latency overhead and 5% throughput overhead over kernel-bypass Redis.
- On the Retwis benchmark, Persimmon incurs no discernible latency overhead and 5%–8% throughput overhead for both standard and kernel-bypass TAPIR.

Furthermore, on gigabyte datasets, both Redis and TAPIR can recover within 15 s after a crash. Porting each application to Persimmon required less than 150 lines of code changes.

Although this paper mainly focuses on porting *existing* in-memory applications to PM, Persimmon also simplifies the development of *future* PM application. Even with high-level libraries, like Intel's PMDK [80], it remains difficult to write PM code that is both fast and correct. In contrast, our PSM abstraction lets programmers write state machine code targeting regular memory, then Persimmon automatically provides persistence while correctly maintaining crash consistency with low overhead. Persimmon thus offers a solution for developing new, high-performance persistent applications as easily as developing in-memory applications.

## 2 Persimmon Overview

This section gives an overview of Persimmon, including its design goals and API, and defines the requirements and guarantees of the persistent state machine model. Persimmon is designed for the x86-64 processor with Intel® Optane™ DC Persistent Memory [46, 108]. We assume an underlying POSIX-based OS due to Persimmon's use of fork; however, the design could be modified for other environments.

### 2.1 Design Goals

We identify three goals for Persimmon's design.

**Minimal Application Changes.** Existing in-memory storage systems are highly optimized for low latency in everything from data structures to memory allocators. To maintain these optimizations and reduce programmer effort, Persimmon's first goal is to minimize changes to existing application code.

**Strong Guarantees.** Reasoning about application state after a crash is difficult for PM applications [61,62]. To simplify applications and ensure crash consistency, Persimmon's second goal is to provide strong and clear persistence guarantees.

**Good Performance.** In-memory storage systems must respond to requests within microseconds, so they cannot afford the high cost of existing persistence mechanisms (e.g., logging to disk). To provide persistence with PM, Persimmon's last goal is to impose less than a microsecond of latency overhead on in-memory systems with no persistence while also providing fast recovery times on the order of seconds.

### 2.2 Persimmon Persistent State Machine Model

Persimmon targets distributed systems deployed within a single datacenter that largely keep their state in memory and offer high-performance RPC processing. We assume the application state needed for recovery can be encapsulated in a *persistent state machine* (PSM) with the following properties:

- *Does not have external dependencies*. The state machine must contain no references to state outside the application process's address space; e.g., it cannot have file descriptors or open sockets.
- *Executes deterministically.* Each operation executes identically every time with no dependence on external inputs (e.g., the current time, random numbers) other than the operation arguments.
- *Has no external side-effects.* State machine operations must perform only computation and memory allocation and de-

**Persimmon Interface**

- `psm_init()` → `bool` - Initialization function; returns true if the application is in recovery.
- `psm_invoke_rw(op)` - Invoke read-write `op` with persistence on the state machine.
- `psm_invoke_ro(op)` - Invoke read-only `op` without persistence on the state machine.

Figure 1: Persistent state machine API implemented by Persimmon.

allocation (e.g., `mmap` and `munmap`). They must not invoke code with side-effects outside the application process (e.g., syscalls other than memory allocation).

These properties are common to state machine abstractions, and are required for correct shadow execution with Persimmon. Similar to replicated state machines (RSMs), persistent state machines require that operations execute sequentially for determinism. Due to the popularity of RSMs in the datacenter, we believe this requirement to be reasonable for many applications. For applications that require concurrency, it may be possible to apply existing techniques developed for RSMs [52, 72]; however, we defer the exploration of these techniques to future work.

### 2.3 Persimmon Persistent State Machine API

Persimmon provides a minimal application programming interface through its user-level library. The Persimmon library presents three functions to applications (Figure 1) that: (1) initialize the persistent state machine, (2) invoke a read-write PSM operation and (3) invoke a read-only operation. We offer the third function as an optimization for RPCs that only inspect state but do not update it, since many in-memory applications have a read-heavy workload. Programmers use the invocation functions to call existing application functions (e.g., execution of Redis commands on Redis data structures). Persimmon directly executes these functions on the PSM, so they must follow the properties laid out above.

An application starts by invoking the `psm_init` function, which returns a flag indicating whether the application has just recovered from a crash. If recovered, the persistent state machine will be returned to its state after the last completed operation. If not in recovery, the application should initialize the state machine by invoking an initialization operation (e.g., creating an empty Redis hash table) with `psm_invoke_rw`. The application can then begin RPC processing. On each RPC, we expect the application to invoke `psm_invoke_rw` if the RPC updates application state that is later needed for recovery. For correct recovery, the application must invoke the PSM operation *before* responding to the RPC. For RPCs that only access application state and do not make updates that must later be recovered (e.g., Redis `GET` operations), the application can use `psm_invoke_ro` for lower overhead.

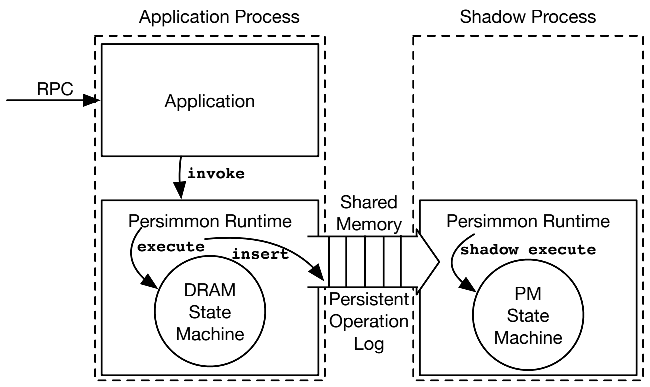


Figure 2: Persimmon runtime.

### 2.4 Persimmon Persistent State Machine Guarantees

Persimmon ensures three guarantees for invoked PSM operations. The first applies to all invoked operations, while the remaining two only apply to `psm_invoke_rw` operations.

- *Linearizability.* Persimmon guarantees that all state machine operations are run in a serial order and that serial order reflects the order in which operations are submitted to Persimmon [40].
- *Durability.* Persimmon guarantees that persistent, read-write state machine operations are never lost once they return, regardless of machine failures. Operations will never roll back and their state modifications are never lost.
- *Failure Atomicity.* Persimmon guarantees that state machine operations are failure-atomic. If the operation has not returned before failure, then on recovery, the state machine will reflect a state entirely before the operation has run or entirely after.

While these guarantees are simple, they are sufficient to build a crash consistent application in the face of failures. Because each state machine operation is failure-atomic, applications can easily maintain crash consistency by grouping updates to related data structures into a single operation and ensuring that no invariants are violated at the end of each operation.

## 3 Persimmon Runtime

Persimmon runs in two processes to support executing unmodified state machine code in DRAM and, for shadow execution, instrumented state machine code on PM. The *application process* runs the application and the DRAM state machine copy, while the *shadow process* runs the crash-consistent, shadow execution of the PM state machine copy. Persimmon's runtime shares the application process' address space with the rest of the application and completely owns the shadow process. Figure 2 summarizes Persimmon's runtime organization.

### 3.1 Data Structures

To ensure fast recovery and failure atomicity for the persistent state machine, Persimmon maintains two in-memory state machine snapshots and the data structures listed in Table 1.

Table 1: Data structures maintained by Persimmon.

| Name | Persistent? | Data structure | Purpose | Elements | Operations |
|---|---|---|---|---|---|
| Operation log | Yes | Fixed-size queue | Records invoked operations | Serialized operations | `push`, `pop` |
| DRAM snapshot | No | — | To execute state machine operations on critical path | — | — |
| PM snapshot | Yes | — | Persists effects of operations | — | — |
| Region table | Yes | Resizable array | Records memory used by PM snapshot | $\langle addr, size, path \rangle$ | `insert`, `remove` |
| Undo log | Yes | Resizable stack | Provides crash consistency for PM snapshot | Data entry: $\langle addr, size, data \rangle$ Commit entry: $\langle new\_tail \rangle$ | `append`, `clear` |

We detail the snapshots and data structures below:

- *Operation log.* Persimmon records each invoked read-write operation in the operation log. The application and shadow processes use the operation log as a shared single producer, single consumer queue.
- *DRAM state machine snapshot.* Persimmon uses this snapshot to execute state machine operations on the critical path. It is always up-to-date and is used to serve all read-only operations without persistence.
- *PM state machine snapshot.* Persimmon asynchronously updates this snapshot using shadow execution. The snapshot is up-to-date up to the end of the log.
- *Region table.* Persimmon records memory allocated by the PM state machine snapshot. The PM snapshot is managed at the granularity of *PM regions*, each of which is a contiguous chunk of PM backed by a file.
- *Undo log.* Persimmon uses write-ahead logging for crash-consistent shadow execution. Persimmon instruments state machine code and record every overwritten memory value in the undo log to ensure that a partially executed state machine operation can be rolled back on recovery.

As Persimmon processes state machine operations, it appends them to the operation log while the shadow process digests the log by re-executing each operation on the PM snapshot. Operations in the log represent how far the PM snapshot lags behind the DRAM snapshot. On recovery, operations in the log must be re-executed on the PM snapshot before the application can restart. We keep the log size below a fixed upper bound to ensure that the PM snapshot does not lag the DRAM snapshot by too much and require too much re-execution on recovery. Persimmon implements the operation log as a circular buffer with head and tail pointers, and assumes no operation's arguments are larger than the log size.

### 3.2 Initialization and Normal Execution

When the application calls `psm_init`, Persimmon initializes its runtime in the following way:

1. Allocate the operation log.
2. Start the shadow process.
3. Initialize the DRAM and PM state machine snapshots.
4. Initialize the region table with PM region metadata (§ 3.4).
5. Allocate the undo log as a persistent array of entries (§ 4.2).

As the application runs, it invokes state machine operations through Persimmon, which are recorded to the log and eventually applied to the PSM. For each operation invoked through `psm_invoke_rw`, Persimmon performs the following:

1. Executes the operation on the DRAM snapshot.
2. Persists the operation as an entry in the operation log;
3. If the operation log is full, blocks until the shadow process digests more operations, freeing up space in the log.
4. Asynchronously, the shadow process re-executes each operation in the log on the PM snapshot using crash-consistent shadow execution (§ 4).

For operations invoked with `psm_invoke_ro`, Persimmon skips Steps 2–4. Persimmon blocks the application if the operation log is full. This design limits recovery time but requires that the shadow execution not lag behind too much as the application runs state machine operations. As a result, if the application invokes too many read-write state machine operations at a time, Persimmon will slow application performance significantly. We explore this phenomenon in our evaluation (§ 7.2.1).

### 3.3 Persimmon Shadow Process

Persimmon uses a separate process to perform shadow execution (the "shadow process"), where it switches to a dynamically instrumented version of the application for running the persistent state machine. Persimmon uses this instrumented version to manage persistent memory and ensure failure atomicity, which we discuss in §§ 3.4 and 4.3, respectively.

During initialization, Persimmon creates the shadow process by using `fork` to create a copy of the application process. Immediately after forking, the shadow process checkpoints itself using an existing Linux process checkpointing tool. This checkpoint conveniently stores essential process state that is orthogonal to Persimmon's main functionality (e.g., the process ID), and serves as a "base image" on which Persimmon manages PM regions. This initialization must happen before the application sets up external dependencies (e.g., opens sockets) to avoid causing process checkpointing to fail.

After taking the checkpoint, Persimmon replaces the shadow process's address space with persistent memory by creating a PM region for each existing application memory region. Specifically, Persimmon iterates through the existing memory regions using Linux's `/proc/self/maps` interface. For each region, Persimmon writes its content to a new PM file, mmaps the file into the address space *over* the existing region, and inserts an entry into the PM region table. We skip over read-only regions, which we assume will never become writable; the stack region, which we assume contains no persistent state (§ 4.3); and the operation log.

Finally, the shadow process begins shadow-executing state machine operations from the operation log. Although it executes the same state machine code as the application process does, the code is executed on the shadow process' persistent address space, and so any modifications to memory are reflected in the PM state machine snapshot. However, Persimmon cannot directly execute unmodified application code on PM because it is not failure atomic and allocates DRAM, not PM. Instead, the shadow process turns on dynamic instrumentation to capture memory allocation and writes in order to allocate PM and write to it in a failure-atomic manner.

### 3.4 Persistent Memory Management

To be able to recover the shadow process after a crash, Persimmon must manage its persistent memory and keep track of its metadata persistently. Persimmon manages the shadow process's persistent memory at the granularity of *PM regions*, each of which is contiguous range of persistent virtual memory. A PM region's content is stored in a file in a direct-access (DAX) file system [59] on persistent memory; the file is mmap'ed into the shadow process' address space, allowing access to PM. Persimmon uses a persistent *region table* to manage metadata for PM regions; each element in the table has the form $\langle addr, size, path \rangle$, denoting a PM region $[addr, addr+size)$ backed by a file located at *path*.

Persimmon keeps the region table in an immutable file in PM. Whenever the region table changes, Persimmon writes the entire updated table to a new file and removes the old. Any PM region files that are "orphaned" after a region table update are garbage-collected after the new region table is written. This mechanism provides failure atomicity for region table updates; although expensive, it is easy to implement and invoked only rarely. Because the table is small, Persimmon keeps a cached copy of it in DRAM.

Every time the shadow state machine allocates or frees memory, Persimmon must translate the operation to allocate or free PM regions instead. Persimmon uses dynamic instrumentation to intercept mmap and munmap system calls, which are typically made by the application's memory allocator.[1] Persimmon's PM management thus operates underneath the memory allocator and does not constrain the application to use one specific allocator.

Persimmon currently supports mmap calls that allocate anonymous memory with no address requirement / hint or backing file. For a mmap call of this type, Persimmon creates a PM region of the allocated length and transparently maps the PM region to the intended address. We currently don't distinguish among page protection bits and assume that all allocated pages have all permissions. Persimmon supports munmap calls that free a single PM region or a part thereof, updating the region table before letting the calls through.

[1]A third memory management system call is brk, which is not supported by our current implementation but can be similarly supported.

## 4 Crash-Consistent Shadow Execution

Persimmon's shadow execution uses dynamic instrumentation and undo logging to provide failure atomicity for state machine operations executing arbitrary application code. We chose dynamic binary instrumentation over static compiler instrumentation because application code often calls functions from dynamically linked external libraries (e.g., string functions in libc), which are only available in binary form at runtime. However, dynamic instrumentation comes with higher overhead than static, and a future direction is to improve instrumentation performance by combining static and dynamic instrumentation [96].

### 4.1 Overview

During shadow execution, Persimmon uses an *undo log* in PM to record the value at a persistent location before it is overwritten, similar to many prior systems [9, 16, 33, 81, 88, 99]. To support arbitrary application code in the PSM, Persimmon uses memory-level physical logging so that it can roll back an incomplete state machine operation at recovery time by copying back the previous memory values. The undo log is a sequence of entries, which come in two types:

- A *data entry* records the old value at a persistent location.
- A *commit entry* signifies that a state machine operation has finished; it contains a sole field `new_tail` recording what the operation log's tail pointer should advance to after the current operation is consumed.

The undo log supports append and clear operations. Each operation blocks until it persists.

In the shadow process, Persimmon instruments every write to a persistent location to append a data entry to the undo log before letting the write through. When a state machine operation completes, we *commit* the operation and remove it from the operation log following these steps:

1. Flush all previous writes and wait for them to persist.
2. Compute the updated tail pointer for the operation log and append a commit entry to the undo log.
3. Remove the operation from the log by advancing the tail pointer as computed.
4. Clear the undo log.

The recovery procedure either finishes committing the operation in progress according to the commit entry if one exists, or rolls it back (§ 5).

Undo logging dominates the shadow state machine's performance because (1) it could add additional work to every memory write, and (2) an undo log append must wait for persistence to PM, which is slow. Therefore, our design aims to reduce the number of undo log appends and the amount of extra code executed per application memory write.

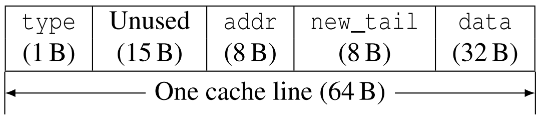


Figure 3: The layout of an undo log element (§ 4.2).

To achieve these goals, Persimmon logs at the granularity of aligned 32 B data blocks. Writes that straddle blocks will generate multiple undo log entries, and writes smaller than 32 B will result in at least 32 B of data being logged. This strategy is motivated by the observation that if a location has been undo logged, then any subsequent writes to that location need not be logged. By logging in larger blocks, Persimmon takes advantage of spatial locality in memory writes to coalesce logging for adjacent locations. Insisting that all blocks be equal-sized and aligned ensures that blocks never overlap and simplifies the detection of duplicate blocks (§ 4.3).

### 4.2 Undo Log Layout and Operations

The undo log consists of a persistent array $A$ of fixed-size 64 B elements (which is the cache line size), and the undo log size $n$, which is stored in DRAM only. Our implementation fixes the array's total size to $2^{20}$ elements (32 MB), which is large enough for our applications. The array $A$ is cache line-aligned, and so are its elements.

An array element either is *valid*, representing an undo log entry (§ 4.1), or is *invalid*. We maintain the invariant that $A[0..n-1]$ contains valid elements and $A[n..]$ contains invalid elements, so that $n$ can be inferred from $A$ upon recovery.

Figure 3 shows the layout of an array element. Each element is interpreted based on its `type` field:

- If `type` $= 0$, the element is invalid.
- If `type` $= 1$, the element is valid and represents a data entry that records the original value of $[\texttt{addr}, \texttt{addr} + 32\,\text{B})$. The `addr` field must be a 32 B-aligned address.
- If `type` $= 2$, the element is valid and represents a commit entry with `new_tail` (§ 4.1).

When appending an entry, we make sure that the `type` field, which also indicates validity, persists no earlier than the other fields. This does not require using an extra persist barrier—since writes to the same cache line reach PM in program order [19, 84], we simply need to write the `type` field last.

To clear the log, we set `type` to zero for elements $A[0..n-1]$ from left to right. If a crash occurs during the clearing, the recovered process will see that $A[0]$ is invalid and will then clear the entire array again.

With this undo log organization, an append requires only one persist barrier (at the end), and consecutive appends write to PM sequentially and avoid repeatedly flushing a single cache line (which is known to incur high latency on Optane persistent memory [12, 92]). Although clearing the log at commit time requires writing to all $n$ entries, it is rare due to the batch commit optimization (§ 4.3) and can be optimized by, e.g., maintaining a persistent commit sequence number.

### 4.3 Dynamic Binary Instrumentation

Persimmon dynamically instruments memory writes for undo logging. The bulk of the logging logic is implemented in the function `log_write(addr, sz)`, which rounds up the range $[\texttt{addr}, \texttt{addr} + \texttt{sz})$ to aligned 32 B blocks and appends an undo log entry for each block. Persimmon, in the shadow process, inserts a call to `log_write` before each application instruction that can write to memory. It translates repeat string operations into regular loops to instrument each iteration separately. For conditionally executed instructions, the instrumentation is executed only when the instruction is.

To minimize overhead from dynamic instrumentation and undo logging, Persimmon applies a number of optimizations:

**Skipping the stack.** Persimmon assumes that the application holds no persistent data on the stack. It does not save stack pages to PM, and as a result it does not need to instrument stack operations. Persimmon assumes that any memory operand that is an offset from the stack pointer `%rsp` points to the stack, and can thus efficiently skip instrumentation for a large number of instructions (notably, all `push`es and `pop`s). This assumption about `%rsp` usage can be validated at runtime by tracking all updates to the register [96], although we have not implemented this validation. The `log_write` function also skips writes to locations above `%rsp` minus 128 B (accounting for the red zone [63]), thereby filtering out any stack operations that do not use an offset from `%rsp`.

**De-duplicating undo log entries.** To avoid logging duplicate data, the `log_write` function maintains a hash set in DRAM for the `addr` field of existing undo log entries, and avoids appending entries whose `addr` already exists. This hash set must support lookup, insert, and clear operations, and fast lookup is key to making this optimization worthwhile.

Our hash set, closely modeled after the CPython dictionary [23], is implemented as a flat array of `addr`'s and resolves collisions with open addressing. The array size is fixed to $2^m$ (where $m = 14$ in our implementation); we use the simple hash function $h(\texttt{addr}) = \texttt{addr}/32$ (since `addr` is aligned to 32 B); and probing uses a linear recurrence with perturbation [23]. A zero element denotes an empty slot, and the hash set is cleared by zeroing out the entire array. Shadow execution commits once the hash set's load factor reaches 50% (see the batch commit optimization below); in case the hash set becomes full, the de-duplication optimization is disabled.

With this hash set implementation, lookups that do not encounter collision are extremely fast. This allows us to insert a fast path de-duplication check, which we detail next.

**Fast-path de-duplication check.** Although de-duplication in `log_write` avoids redundant logging, the function call before every write is still costly as it requires saving and restoring application registers. We therefore insert a "fast path" check *before* the function call to filter out easy-to-identify duplicates. For a memory write to address *dest* of size *sz*, the call to `log_write` is skipped if both conditions hold:

Listing 1: Instrumentation inserted before a memory write of $sz$ bytes. `%dest` and `%tmp` are placeholders for any two distinct general-purpose 64-bit registers. The `.hash_array` label refers to the base address for the hash set array.

```
    (Reserve registers %dest and %tmp)
    (Compute destination of the write, store in %dest)
    (Reserve arithmetic flags)

    # First check: the write is contained in
    # a single block (not generated if sz=1).
    leaq    (sz−1)(%dest), %tmp
    xorq    %dest, %tmp
    cmpq    $31, %tmp
    ja      .slow_path

    # Second check: look in the hash set.
    movq    %dest, %tmp
    shrq    $2, %tmp
    andl    $131064, %tmp
    movq    .hash_array(%tmp), %tmp
    xorq    %dest, %tmp
    cmpq    $32, %tmp
    jb      .skip

  .slow_path
    (Save application registers)
    (Call log_write)
    (Restore application registers)

  .skip
    (Restore arithmetic flags)
    (Restore registers %dest and %tmp)
```

- The write is contained in a *single* aligned 32 B block, i.e.,
$$\lfloor dest/32 \rfloor = \lfloor (dest + sz - 1)/32 \rfloor.$$
- The block's address is found in the hash set on first try (without any probing), i.e., $H[i] = addr$ where $H$ is the hash set array, $i = \lfloor dest/32 \rfloor \bmod 2^m$ according to the hash function, and $addr = \lfloor dest/32 \rfloor \times 32$ is the block address.[2]

Any memory write filtered out by the check is guaranteed to be a duplicate, and the check proves effective in our evaluation (§§ 7.2.1 and 7.2.3). Furthermore, as the computation required by the checks only require bit manipulations, the two checks can be implemented in 11 instructions using only two extra registers (one of which stores *dest* and is anyway required). Listing 1 shows the code inserted before a memory write.

**Batch commit.** Persimmon shadow-executes multiple state machine operations before committing, thus avoiding duplicate logging across multiple operations. After executing each operation, Persimmon checks to see if the de-duplication hash set is more than 50% full and if so, commits. We defer more intelligent batch sizing to future work.

**Skipping newly allocated regions.** Writes to a PM region that is newly allocated (i.e., after the most recent commit) do not need to be logged since, in case of a crash, the state machine will be reverted to before the region was allocated. The `log_write` function therefore maintains, in DRAM, a list of address ranges for newly allocated regions and searches through the list to skip logging such writes. This optimization is critical for supporting `calloc` implementations that manually zero out pages allocated with `mmap`.

[2]Note that if no element exists in the hash set with hash value $h(addr)$, we have $H[i] = 0$ and the check fails as expected (assuming that the application never writes to the block starting at address zero).

## 5 Recovery

To minimize recovery times, recovery in Persimmon is relatively simple. After a crash, the first step is to restore the PM state machine snapshot to a consistent state:

- If the undo log contains a commit record, we set the operation log tail to *new_tail*. If an updated region table exists in PM, we switch to it and garbage collect the old region table. This completes the commit.
- If the undo log contains no commit record, we delete and garbage collect the updated region table (if one exists) and copy the old values from the undo log back to their respective locations. This rolls back the operation in progress at the time of the crash.

As a last step, we clear the undo log in both cases.

Starting from the consistent PM snapshot, Persimmon digests any remaining operations in the operation log so that all previously invoked operations are reflected in the snapshot. Replaying the log ensures that Persimmon maintains its guarantee that any invoked operation that returns will not be lost. This up-to-date snapshot is then copied into DRAM for the application process, and the application is restarted. Assuming that the PSM has captured all persistent application state, the application should be back in its pre-crashed state.

## 6 Implementation

We have implemented Persimmon in C++; it targets x86-64 Linux applications written in C or C++. We use CRIU [24] (v3.12) for process checkpointing during background process initialization (§ 3.3). For dynamic instrumentation (§ 4.3), we use DynamoRIO [8], a runtime code manipulation system. Persimmon's DynamoRIO client is linked into the application along with the DynamoRIO runtime. This setup allows Persimmon to start the application uninstrumented and only begin instrumentation in the background process once it is forked off. To avoid interfering with the application, our instrumentation code takes care not to call into shared libraries (e.g., libc), and instead uses DynamoRIO's memory allocator and our custom system call wrappers.[3]

## 7 Evaluation

Using our implementation, we demonstrate that Persimmon:

- Requires only a small amount of code modification for distributed in-memory storage systems.
- Achieves low overhead on workloads compared with no persistence for both Linux and kernel-bypass applications.
- Recovery quickly even for large memory sizes.

[3]Because our DynamoRIO client is linked into the application, it is not loaded by DynamoRIO's private loader, which would have created a separate copy of each library used by the client.

Table 2: Rough lines of code changed to port Redis and TAPIR.

| | Lines added / changed | |
|---|---|---|
| | Redis | TAPIR |
| Initialize Persimmon | 7 | 10 |
| Factor out state machine init. | 36 | 34 |
| Serialize state machine operation | 26 | 12 |
| Deserialize & execute operation | 45 | 25 |
| Check for read-only operations | 1 | 1 |
| Refactor for better performance | — | 57 |
| **Total** | 115 | 139 |

We ported Redis, a popular key-value store, and TAPIR [113], a distributed transactional data store, to use Persimmon. We first describe the code changes required to port these applications (§ 7.1), followed by performance comparisons (§§ 7.2.1 and 7.2.2). Finally, we use microbenchmarks to evaluate the effectiveness of Persimmon's optimizations (§ 7.2.3).

## 7.1 Programming Experience

Although the Persimmon API is simple (Figure 1), porting real applications can require a few extra steps as real-world code bases are not always well-organized into a state machine abstraction, even if the application is processing RPCs. The programmer typically needs to:

1. Add a call to `psm_init()`, passing configuration arguments like the PM file system location.
2. Factor out the state machine initialization code into a single function, separating it from other initialization (e.g., network I/O), so that it can be skipped on recovery.
3. Write a function that serializes a state machine operation for operation logging. One can reuse the application's existing RPC serialization, but, for better performance, we found it valuable to use a custom format that is cheaper to parse in the shadow process.
4. Write a function that deserializes and executes an operation, to be invoked under instrumentation in Persimmon's shadow process. This function typically only needs to call the application's RPC handler using the deserialized operation, but may need to suppress any I/O by the handler (e.g., sending a reply over the network).
5. Insert checks to distinguish read-write operations from read-only ones; such checks likely already exist for applications that support state machine replication. Invoke Persimmon for read-write operations.

We performed all of these steps for Redis because it was not well-organized into a state machine, especially because it is written in C, which is not an object-oriented language. TAPIR, on the other hand, was is already designed as a state machine to work with its replication mechanism. Table 2 summarizes the code changes required to port Redis and TAPIR. We discuss each application in detail next.

### 7.1.1 Porting Redis

To port Redis, we treat each Redis command as a state machine operation and invoke it through Persimmon. To summarize, the changes made were:

1. Persimmon initialization took 6 lines of code (LoC).
2. To factor out the state machine initialization code, we separated out three blocks of code ($7+7+22$ lines) responsible for network and domain socket initialization, etc.
3. Redis operation serialization for the state machine took 26 lines of code. The serialization consists of the address of the Redis command's handler function[4] as well as the command's arguments.
4. Redis operation deserialization, parsing, and dispatch took 24 lines of code. Since executing Redis commands requires a "client", we reuse the fake client code from Redis AOF (21 lines), which also suppresses replies.
5. To determine whether an operation is read-only, we reused existing code from Redis' state machine replication.

In all, Persimmon allowed us to achieve persistence for Redis with roughly 100 lines of code changes. In contrast, Redis' own persistence implementation (AOF and RDB) consists of roughly 3000 lines of code, including complex logic such as request processing while log compaction is in progress. As another point of comparison, Pmem-Redis [82], a version of Redis that uses persistent memory, contains roughly 30 000 new lines of C code over Redis 4.0.0, although we note that Pmem-Redis contains features that are orthogonal to Persimmon (e.g., defragmentation). Xu et al. report that manually porting Redis to persistent memory using PMDK [80] "is not straightforward and requires large engineering effort" [104, § 3.3]. They list five difficulties, which include supporting the many different Redis objects with different encodings, carefully ordering writes to maintain crash consistency, etc. None of the difficulties arose when we ported Redis using Persimmon.

Despite minimal changes, our port is the most feature-complete PM Redis port that we know of. For example, Persimmon-Redis supports complex Redis data structures like sets and hashes while the persistent Redis from WHISPER [73,98] only supports simple key-value pairs, and Pmem-Redis lacks support for optimized encodings like `intset` and `zipmap`. We also support hash table resizing, which is not supported by P-Redis due to its complexity [104]. Persimmon-Redis supports these features "out of the box", without requiring additional code for each feature. In addition, Persimmon lets Redis retain jemalloc [47] as its memory allocator, which was carefully chosen by the Redis developers [86].

### 7.1.2 Porting TAPIR

The TAPIR transactional data store is built on top of the inconsistent replication (IR) protocol. We treat each IR RPC as a state machine operation. In the application, these RPCs

[4] As the shadow process is forked from the application process (§ 3.3), the application's code segment is mapped at the same location in both processes.

are serialized using Protocol Buffers [83], and a naive Persimmon port uses the same serialization for operation logging. Although easy to implement, this strategy causes significant performance degradation as Persimmon has to execute Protocol Buffer parsing under shadow execution, which is slow.

To improve performance, we refactored TAPIR's IR RPC handlers to take individual RPC fields as arguments, so that they can be called from the shadow process without first constructing protobuf objects. This refactoring involved roughly 50 LoC changes, which were mostly mechanical, and enabled operation logging using a simple custom format (like for Redis). In addition to this refectoring, we did the following:

1. Persimmon initialization took 11 lines of code.
2. To factor out state machine initialization, we moved two code blocks ($10+4$ lines) and modified ~20 LoC to allow creating a RPC handler without a network connection.
3. Operation serialization took 12 lines of code.
4. Operation deserialization and invocation took 17 lines of code, plus an extra 8 lines to suppress replies.
5. We reused application code to detect read-only operations.

The overall code change amounts to roughly 140 lines in total.

To recover from replica failures, TAPIR uses a complex in-memory recovery protocol, inspired by VR, which has been proven to be incorrect [69]—under certain failure conditions, TAPIR can lose operations when recovering, causing it to miss updates. Persimmon-TAPIR fixes this problem transparently and lets replicas correctly recover their state on failures.

One of the benefits of TAPIR is that replicas can recover and immediately begin processing transactions. However, without the most up-to-date state, these recovered replicas will degrade performance by serving stale reads and unnecessarily aborting writes. TAPIR particularly suffers from this performance degradation because it needs $\frac{3}{2}f+1$ to use the single-round trip fast path. For a 3-machine replica group, this number includes all of the participants. As a result, while the recovering replica is able to participate in transactions immediately, without the most up-to-date state, it is only hurting performance. However, Persimmon-TAPIR can significantly reduce this performance degradation by limiting the amount of state that the replica needs to recover.

### 7.2 Performance Evaluation

We evaluate Persimmon on three 52-core, dual-socket Intel Xeon Platinum 8272 2.6 GHz servers, each with 3 TB of Intel® Optane™ DC Persistent Memory in app direct mode and 768 GB of DRAM. We mount an ext4 file system in DAX mode [59] on the PM. Persimmon's application and background processes run on two physical cores on a single NUMA socket and use only DRAM, PM, and the NIC on that socket. To supply the client workload, we use a server with a 20-core dual-socket Xeon Silver 4114 2.2 GHz CPU, connected with with Mellanox CX-5 100 Gbps NICs and an Arista 7060CX 100 Gbps top-of-rack switch.

Table 3: Summary of Redis performance on YCSB (Zipfian constant $=0.75$, 10% update, median of five runs). The persistence options are volatile ("Vol"), persistent through Persimmon ("PSM"), and persistent through append-only file ("AOF"), with our performance in bold. Shown are median latency at low load and peak throughput. Persimmon incurs, for Linux, negligible latency and throughput overhead and, for kernel bypass, negligible latency overhead and ~5% throughput overhead. Figure 4a shows the full latency vs throughput graph.

| | **Latency** (µs) | | | **Throughput** (Kops) | | |
|---|---|---|---|---|---|---|
| **Redis Setup** | Vol | **PSM** | AOF | Vol | **PSM** | AOF |
| Linux | 17.8 | **17.5** | 18.6 | 227 | **227** | 107 |
| Bypass | 10.4 | **11.2** | 12.0 | 452 | **429** | 130 |

#### 7.2.1 Redis Performance

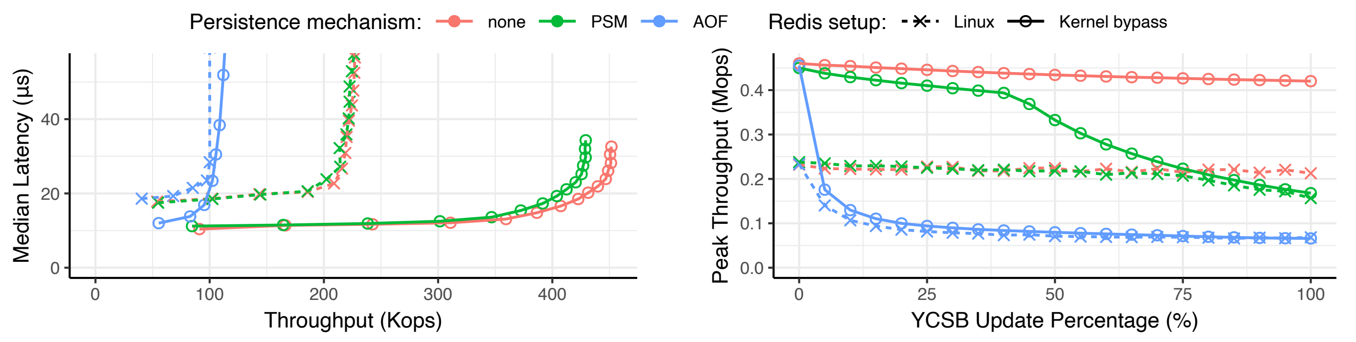
We use Persimmon to add persistence to two versions of Redis—regular Redis (v4.0.9), which processes requests over TCP using the POSIX API, and a high-performance, kernel-bypass version of Redis that uses the Demikernel's DPDK library OS [26, 112]. We compare both Persimmon-Redis versions to Redis's existing persistence mechanisms.

While Redis is an in-memory storage system, persisting Redis is popular enough for it to integrate two mechanisms for saving its state [85]: RDB, a snapshotting mechanism, and AOF, an append-only operation log. Using RDB requires pausing operation processing for a short period while Redis spawns a background process to checkpoint its database.

AOF logs every write operation received by the server to a file, similar to Persimmon's operation logging. However, Redis typically recommends only `fsync`'ing those logged operations periodically to avoid performance overhead [85], so operations can be lost on a crash. AOF must also avoid the operation log growing unboundedly, so it periodically creates an RDB snapshot, letting it truncate the log. This process further degrades performance, so the Redis developers recommend only snapshotting once or twice an hour [85].

In some sense, these mechanisms are orthogonal to Persimmon. They have features that Persimmon does not provide (e.g., compact and platform-independent serialization), but do not provide cheap, general-purpose persistence to in-memory storage systems. Their implementation is specific to Redis, requires significant implementation effort, and, as shown below, can cause large performance degradation.

**Experiment setup.** We evaluate Redis performance using the YCSB benchmark [21]. We implemented a custom multi-threaded YCSB client, using Shenango [78], which supports both TCP and Demikernel's UDP-based protocol. Following the official YCSB implementation [111], our client is closed-loop and does not use Redis pipelining, each YCSB record is represented with a Redis hash, and fields are accessed using Redis' `HSET`/`HGET` commands. We load 13 million records; as is YCSB default, each record has 10 fields (i.e., 130 million items in total), and each field has a 100 B value. Our client is-



(a) *Latency vs throughput for a 10%-update workload.* Persimmon incurs negligible overhead over the Linux baseline, and a roughly 5% overhead to the peak throughput over the kernel-bypass baseline.

(b) *Peak throughput vs update percentage.* Persimmon throughput stays within 9% of the kernel bypass baseline for up to 40%-update, and within 4% of the Linux baseline for up to 75%-update.

Figure 4: Redis performance on the YCSB benchmark (median of five runs, Zipfian constant = 0.75).

sues reads and updates according to a fixed ratio, and chooses which records to access according to a Zipfian distribution.

On the server side, the regular Redis server uses jemalloc (as is recommended for Linux) and our kernel-bypass Redis uses the Hoard memory allocator [3] (following the Demikernel implementation [26]). Where Redis AOF is enabled, we place the AOF log file on the PM file system, disable AOF rewriting, and configure it to always `fsync` before sending replies, providing the same level of durability as Persimmon.

**End-to-end performance.** We start with the end-to-end performance for a typical YCSB workload with 10% updates and a Zipfian constant of 0.75. We measure the latency and throughput for unmodified Redis, Persimmon Redis, and Redis AOF. Table 3 reports performance both on Linux and with kernel-bypass enabled through the Demikernel and shows:

- On Linux, Persimmon provides persistence while incurring negligible latency or throughput overhead.
- With kernel bypass, Persimmon incurs negligible latency overhead at low load, and a 5% degradation to peak throughput. Kernel bypass makes Redis significantly more efficient, so Persimmon has slightly more impact on performance.
- On Linux, AOF incurs $< 1\,\mu$s latency cost but a $2\times$ throughput penalty, a much higher overhead than Persimmon.
- With kernel bypass, AOF again incurs a small latency overhead but a $3.5\times$ throughput loss.

Some of the overhead of AOF is likely due to inefficiencies in accessing PM through ext4 and could be reduced using a specialized PM file system like NOVA [104, 105] or SplitFS [48]. Overall, Persimmon offers persistence at a much lower cost than Redis's own custom persistence mechanism both on Linux and for future kernel-bypass deployments.

Figure 4a shows the full latency vs throughput plot for this workload with varied numbers of closed-loop clients. (Lower and to the right is better. The knee where latency goes up shows the peak throughput.)

Table 4: Redis recovery time and storage size for the three persistence mechanisms (median of three runs). Persimmon recovers $4.6\times$–$6\times$ faster than AOF and RDB. The discrepancy between the Linux and kernel bypass implementations is likely due to memory allocators differences (jemalloc vs Hoard).

| | **Recovery Time** (s) | | | **Storage Size** (GB) | | |
|---|---|---|---|---|---|---|
| **Redis Setup** | **PSM** | AOF | RDB | **PSM** | AOF | RDB |
| Linux | **14.6** | 87.4 | 87.8 | **23** | 16 | 2.8 |
| Bypass | **20.4** | 93.3 | 93.5 | **33** | 16 | 2.8 |

**Peak throughput vs update ratio.** Since Persimmon only logs and shadow-executes write operations, its overhead depends on the workload's update-to-read ratio. Figure 4b shows peak Redis throughput as we vary the workload's update percentage. Persimmon's throughput remains within 4% of the baseline for up to 75%-update on Linux (represented by the red and green "×" lines in the middle of the graph), and within 9% of the baseline for up to 40%-update for kernel bypass (represented by the red and green "○" lines at the top). After these points, the shadow state machine becomes saturated and the throughput drops precipitously. Both versions can easily handle read-heavy workloads, which are common in practice.

**Recovery speed and storage size.** Table 4 shows Redis' recovery speed and storage usage under different persistence mechanisms if we kill Redis after loading our YCSB dataset.[5] Because Persimmon persists application data in its in-memory format, the bulk of its recovery is physically copying PM regions back into DRAM, while AOF and RDB require reloading the database. Consequently, Persimmon recovery is faster by $4.6\times$ (on Linux) to $6.0\times$ (for kernel bypass) and, we believe, can be further optimized by copying PM regions in parallel using multiple cores. However, Persimmon's space

[5]The Persimmon recovery measurements do not include operation replay because it would take negligible time—since the operation log is only 32 MB, replaying even a full log would only take roughly one second for YCSB.

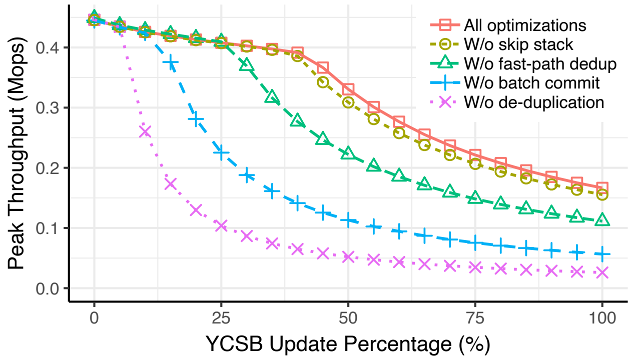


Figure 5: Disabling each optimization degrades Redis throughput on YCSB (Zipf = 0.75, kernel bypass, median of five runs). Note that disabling de-duplication also disables the fast-path check.

usage tracks the application's RAM usage while AOF and RDB can use more compact serialization formats. The discrepancy between Linux and kernel-bypass Redis is most likely because they use different memory allocators (jemalloc vs Hoard), which lead to differing amounts of memory consumption and exhibit different allocation performance.

**Effectiveness of optimizations.** To evaluate Persimmon's optimizations (§ 4.3), we measure Redis performance under Persimmon after disabling each optimization separately (note that disabling undo log de-duplication also disables the fast-path de-duplication check).[6] Figure 5 shows that no optimization can be removed without degrading performance.

### 7.2.2 TAPIR Performance

As with Redis, we use Persimmon to add persistence to two versions of TAPIR—regular TAPIR, which processes requests over UDP using the POSIX API, and kernel-bypass TAPIR, which uses the Demikernel's DPDK library OS. We compare each version to the original, non-persistent application.

We evaluate TAPIR performance using the Retwis benchmark [113], a Twitter-like transactional workload, with 10 million keys (where keys and values are 64 B) and a Zipf coefficient of 0.75. On the server side, we configure one shard with three replicas running on separate machines equipped with PM. For clients, we use a multi-process closed-loop load generator that processes RPCs over UDP using the POSIX API; it supports both the regular TAPIR and Demikernel protocols.

Table 5 reports the mean latency and peak throughput for Retwis transactions. Persimmon incurs negligible latency overhead for both the Linux and the kernel-bypass setups. For peak transaction throughput, Persimmon incurs a 5.4% degradation on Linux and a 7.3% degradation for kernel bypass. Note that the TAPIR transaction latency is much higher than the Redis latency from § 7.2.1 because each transaction in-

[6]We excluded the optimization that skips newly allocated regions because no new regions are allocated during these experiments (since our workload only overwrites existing keys). However, this optimization helps tremendously when loading the initial YCSB database.

Table 5: Summary of TAPIR performance on Retwis (Zipf = 0.75, three replicas). The persistence options are volatile and Persimmon ("PSM"), with our performance in bold. Shown are the mean transaction latency at low load (measured using five clients) and peak throughput (with at most 20 clients). Persimmon incurs negligible latency overhead and a 5%–8% throughput overhead. Figure 6 shows the full latency vs throughput graph.

| | **Latency** (µs) | | **Throughput** (txn/s) | |
|---|---|---|---|---|
| **TAPIR Server** | Volatile | **PSM** | Volatile | **PSM** |
| Linux | 342 | **338** | 37 K | **35 K** |
| Bypass | 310 | **309** | 41 K | **38 K** |

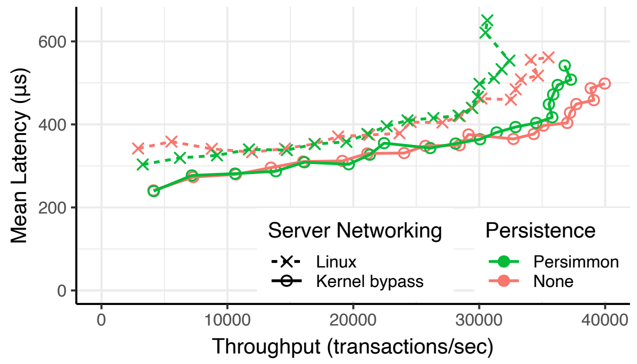


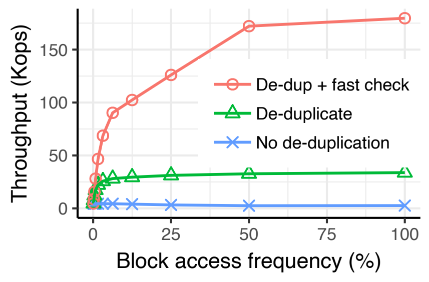
Figure 6: Latency vs throughput for TAPIR on the Retwis benchmark (10 million keys, Zipf = 0.75, median of five runs). Throughput starts decreasing as more clients are added due to congestion collapse.

volves multiple RPCs sent to three replicas; and kernel bypass provides less benefit for TAPIR than for Redis because TAPIR performs more work per RPC and its code base is less heavily optimized. Figure 6 shows the full throughput vs latency plot for this workload (lower and to the right is better). After a crash, Persimmon can recover a replica within 7 s; we were not able to compare to a recovery baseline as TAPIR has not implemented recovery from another replica.

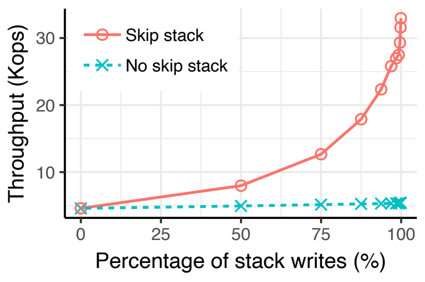
### 7.2.3 Optimization Microbenchmarks

We use microbenchmarks to demonstrate the effectiveness of Persimmon's optimizations for crash-consistent shadow execution (§ 4.3). Each microbenchmark repeatedly invokes operations using `psm_invoke_rw`. Each operation performs 1024 memory accesses, where each access reads a 32 B *block* of memory into a `%ymm` register, performs an AVX-2 vector addition on it, and writes it back to the same memory location. We picked the access size of 32 B to match the undo logging granularity (§ 4.2). For clarity, we turn off batch commit unless otherwise specified. In each benchmark, we disable certain optimization(s) and use a memory access location pattern that demonstrates the effect of the optimization(s).

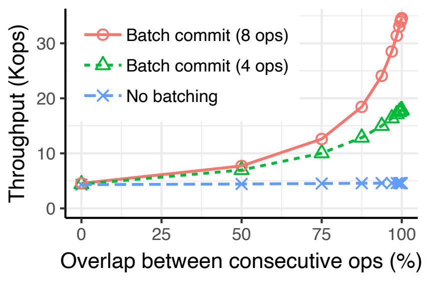
In Figure 7a, we disable undo log de-duplication and its fast-path check and vary the access frequency of each block—ranging from each operation accessing 1024 different



(a) *Undo log de-duplication* is most effective when few locations are accessed repeatedly.



(b) *Skipping stack operations* is most effective with a large percentage of stack accesses.



(c) *Batch commit* is most effective when operations access a lot of overlapping memory.

Figure 7: Effectiveness of Persimmon's optimizations (median of five runs).

blocks sequentially to accessing a single block 1024 times. When memory accesses are concentrated on few locations, de-duplication can deliver up to 13× throughput increase, and the fast-path check, an additional 5×. These optimizations incur no discernible overhead when there is no duplication.

In Figure 7b, we disable the optimization that skips stack accesses and direct a percentage of memory accesses to an array allocated on the stack. As expected, this optimization is most effective when state machine operations frequently access the stack, which we assume to be not persistent.

In Figure 7c, we enable the batch commit optimization with fixed batch sizes of 4 or 8 operations, and vary the percentage of overlap between blocks accessed by consecutive operations. Batch commit is most effective when it can group together many operations that access common memory locations.

## 8 Related Work

**PM frameworks.** Because PM's low level interface (load, store, flush) can be hard to use, prior works have proposed PM frameworks that provide higher-level APIs [9,16,18,22,31–33, 41,42,45,60,65,67,81,88,95,99,102,114]. Such a framework typically requires the programmer to (1) explicitly declare persistent data (e.g., by using a special `malloc`), (2) delineate failure-atomic regions using begin/end annotations, and (3) annotate operations that modify persistent data. The framework can then provide durability and failure atomicity by executing extra logic for each persistent operation, e.g., to log the operation and flush any modified persistent locations. Some of these frameworks reduce the programming burden by, e.g., inferring failure-atomic regions from existing synchronization points [9, 33, 41, 45, 60, 102], and by automatically interposing on persistent operations using language features [22, 95], static compiler instrumentation [9, 32, 33, 42, 102], or runtime methods [41, 88, 102].

Using these frameworks comes with two difficulties. First, despite their high-level APIs, porting an application using these frameworks can still be labor-intensive [66, 104] and bug-prone [61, 62]; we elaborate on the porting experience later in this section. Second, PM accesses and failure atomicity mechanisms on the critical path can impose significant performance overhead, especially for frameworks that implement automatic interposition. For example, microbenchmarks by Hsu et al. [41, § 5.6] show a 5×–25× slowdown for NVthreads and a 70×–200× slowdown for Mnemosyne and Atlas on memory accesses. While such overhead might be acceptable for I/O-bound applications, it can significantly slow down high-performance data stores with fast I/O.

Persimmon alleviates both difficulties. On the programming effort side, Persimmon does not require manually annotating of every persistent allocation; it simply persists all state encapsulated by the state machine. We are not aware of any other framework of its kind that provides this feature.[7] Furthermore, by taking a full-process approach to persisting the shadow process machine, Persimmon rules out referential integrity bugs [16] (e.g., arising from PM-to-DRAM pointers) and relocation issues (where a persistent region is mapped to a different address after recovery).

For performance, Persimmon keeps the critical path execution as close to the original application as possible. As far as we know, Persimmon is the only system that can transparently retain the application's memory allocator (rather than swap in a special PM allocator), thus preserving its memory layout.[8] It also pushes all PM accesses (besides operation logging) and instrumentation into the background; this requires an extra CPU core, but minimizes overhead on the critical path.

Finally, Persimmon is the first system to use *dynamic binary instrumentation* to provide failure atomicity on PM. This allows application code to call into libraries that are dynamically linked or whose source is not available (§ 4).

**Porting data structures to PM.** Despite the high-level APIs provided by the PM frameworks, porting existing

[7]Except AutoPersist [88], a Java PM framework that only requires marking a "durable root" object from which all persistent state can be reached. However, it exploits properties of Java / the JVM and cannot be easily extended to support C/C++ applications.

[8]Romulus [22] can be used with any memory allocator but requires manual modification to the allocator code.

data structures to PM remains challenging.[9] For example, Marathe et al. called their experience porting `memcached` "surprisingly non-trivial" [66], and Xu et al. reported five difficulties in porting Redis' hash table using PMDK, e.g., supporting Redis' many object encodings and having to order persistent writes carefully [104, §3.3]. With the high manual effort, bugs are likely to appear in PM code even when using high-level PM frameworks [61, 62]. In contrast, Persimmon requires minimal code changes to port an application (§ 7.1); in particular, we encountered none of the five difficulties identified by Xu et al. as we ported Redis using Persimmon.

Other works have noted that certain classes of data structures are easier to convert to PM and proposed techniques accordingly. For example, RECIPE [57] observes that concurrent indexes that implement helping and non-blocking reads are "inherently crash-consistent"; MOD [39] notes that purely functional data structures can be easily made failure-atomic through copy-on-write; and Friedman et al. [30] automatically transform a special class of lock-free data structures to be persistent by exploiting the *traversal* phase in these data structures' operations. Persimmon makes no such assumptions on the application's data structures.

Like Persimmon, Pronto [68] relies on state machine-like assumptions on the data structure—that it is encapsulated and has deterministic operations. This enables Pronto to implement persistence using semantic logging and periodic snapshotting. In contrast, Persimmon allows porting an entire application, rather than a single data structure, and maintains low latency even for large data sizes as it avoids the periodic stalls from the synchronous phase of snapshotting.

**PM data structures / stores.** Many prior works have redesigned in-memory data structures to be durable in PM. These include both tree-based [1, 10, 11, 14, 17, 44, 53, 56, 93, 97, 109] and hash-based structures [13, 25, 74, 87, 116–118]. There have also been hybrid designs that combine PM with DRAM [12, 43, 77, 100, 103, 110] and/or with SSD [49, 51, 110]. To achieve high performance, these data structures often use data layouts and operations specifically optimized for PM.

In contrast, Persimmon is not one data structure/store; rather, it allows porting an in-memory storage system to be persistent on PM. Although a Persimmon-transformed system might not achieve resource utilization on par with handcrafted PM data stores, Persimmon is more general and can deliver good performance on real-life workloads.

Persimmon's design is similar to that of Bullet [43], a persistent key-value store that serves requests from a "front-end cache" in DRAM, records operations in persistent logs, and uses background threads to apply logged operations to a persistent hash table. While Bullet only supports a limited set of operations, Persimmon generalizes the design to support general application-level operations. A future direction is to incorporate Bullet's cross-referencing logs into Persimmon to better support multi-core applications.

[9] Here we focus on the *porting* experience and thus do not include works that *replace* an application's data structure with a persistent one (e.g., swapping out the hash table of Redis with a persistent B-tree [93]).

**PM-aware file systems.** A natural way of using PM is to view it as a fast storage device and incorporate it into the storage stack. Many file systems have been designed to effectively exploit the high performance of PM [15, 20, 27–29, 48, 50, 55, 64, 94, 101, 105–107, 115]. These PM-aware file systems can transparently speed up durable applications that perform I/O using the file system interface, but do not directly apply to in-memory applications that do not use the file system.

**Logging for crash consistency.** Logging has been used to implement crash consistency in many contexts outside of PM, e.g., in database management systems [4, 5, 34–37, 54, 70] and journaling file systems [6, 38, 89–91]. Persimmon's logging mechanisms are conceptually similar, with a key difference being that we perform undo logging on application-supplied state machine operations (in x86-64) as opposed to SQL transactions or file system operations.

## 9 Conclusion

Persistent memory (PM) offers a promising solution to providing crash recovery to in-memory storage systems. However, manually porting applications to PM remains challenging. Persimmon leverages PM to provide persistence to existing in-memory storage systems while maintaining high performance and requiring minimal code changes. We use Persimmon to add persistence to Redis and TAPIR with ease while incurring minimal performance overhead on common workloads.

## Acknowledgments

We thank our shepherd Vijay Chidambaram and the anonymous reviewers for their helpful comments on the paper. We'd also like to thank Aurojit Panda, David Culler, Vivian Fang, Amy Ousterhout, and other members of the UC Berkeley NetSys Lab, the RISELab, and the Microsoft Systems Research Group for their feedback. This work was funded in part by NSF Grants 1817115, 1817116, and 1704941, and by grants from Intel, VMware, Ericsson, Futurewei, and Cisco.

## References

[1] ARULRAJ, J., LEVANDOSKI, J., MINHAS, U. F., AND LARSON, P.-A. BzTree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow. 11*, 5 (Jan. 2018), 553–565.

[2] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), ACM, pp. 53–64.

[3] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. of ASPLOS* (2000).

[4] BERNSTEIN, P. A., GOODMAN, N., AND HADZILACOS, V. Recovery algorithms for database systems. In *Proceedings of the IFIP 9th World Computer Congress* (1983).

[5] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency control and recovery in database systems*, vol. 370. Addison-wesley Reading, 1987.

[6] BEST, S. JFS log: How the journaled file system performs logging. In *Annual Linux Showcase & Conference* (2000).

[7] BORNHOLT, J., KAUFMANN, A., LI, J., KRISHNAMURTHY, A., TORLAK, E., AND WANG, X. Specifying and checking file system crash-consistency models. In *Proc. of ASPLOS* (2016), pp. 83–98.

[8] BRUENING, D., ZHAO, Q., AND AMARASINGHE, S. Transparent dynamic instrumentation. In *Proc. of VEE* (2012).

[9] CHAKRABARTI, D. R., BOEHM, H.-J., AND BHANDARI, K. Atlas: Leveraging locks for non-volatile memory consistency. In *Proc. of OOPSLA* (2014).

[10] CHEN, S., GIBBONS, P. B., AND NATH, S. Rethinking database algorithms for phase change memory. In *Proc. of CIDR* (2011).

[11] CHEN, S., AND JIN, Q. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow. 8*, 7 (Feb. 2015), 786–797.

[12] CHEN, Y., LU, Y., YANG, F., WANG, Q., WANG, Y., AND SHU, J. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proc. of ASPLOS* (2020).

[13] CHEN, Z., HUANG, Y., DING, B., AND ZUO, P. Lock-free concurrent level hashing for persistent memory. In *Proc. of USENIX ATC* (2020).

[14] CHI, P., LEE, W.-C., AND XIE, Y. Making B+-tree efficient in PCM-based main memory. In *Proc. of ISLPED* (2014).

[15] CHOI, J., HONG, J., KWON, Y., AND HAN, H. Libnvmmio: Reconstructing software IO path with failure-atomic memory-mapped interface. In *Proc. of USENIX ATC* (2020).

[16] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of ASPLOS* (2011).

[17] COHEN, N., AKSUN, D. T., AVNI, H., AND LARUS, J. R. Fine-grain checkpointing with in-cache-line logging. In *Proc. of ASPLOS* (2019).

[18] COHEN, N., AKSUN, D. T., AND LARUS, J. R. Object-oriented recovery for non-volatile memory. *Proc. ACM Program. Lang. 2*, OOPSLA (Oct. 2018).

[19] COHEN, N., FRIEDMAN, M., AND LARUS, J. R. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proc. ACM Program. Lang. 1*, OOPSLA (Oct. 2017).

[20] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proc. of SOSP* (2009).

[21] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proc. of SOCC* (2010).

[22] CORREIA, A., FELBER, P., AND RAMALHETE, P. Romulus: Efficient algorithms for persistent transactional memory. In *Proc. of SPAA* (2018).

[23] cpython: 52f68c95e025 objects/dictobject.c. https://hg.python.org/cpython/file/52f68c95e025/Objects/dictobject.c#l33.

[24] CRIU. https://criu.org.

[25] DEBNATH, B., HAGHDOOST, A., KADAV, A., KHATIB, M. G., AND UNGUREANU, C. Revisiting hash table design for phase change memory. In *Proc. of INFLOW* (2015).

[26] demikernel/demikernel: Demikernel OS. https://github.com/demikernel/demikernel.

[27] DONG, M., BU, H., YI, J., DONG, B., AND CHEN, H. Performance and protection in the ZoFS user-space NVM file system. In *Proc. of SOSP* (2019).

[28] DONG, M., AND CHEN, H. Soft updates made simple and fast on non-volatile memory. In *Proc. of USENIX ATC* (2017).

[29] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proc. of EuroSys* (2014).

[30] FRIEDMAN, M., BEN-DAVID, N., WEI, Y., BLELLOCH, G. E., AND PETRANK, E. NVTraverse: In NVRAM data structures, the destination is more important than the journey. In *Proc. of PLDI* (2020).

[31] Genç, K., Bond, M. D., and Xu, G. H. Crafty: Efficient, HTM-compatible persistent transactions. In *Proc. of PLDI* (2020).

[32] George, J. S., Verma, M., Venkatasubramanian, R., and Subrahmanyam, P. go-pmem: Native support for programming persistent memory in Go. In *Proc. of USENIX ATC* (2020).

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

[34] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I. The recovery manager of the System R database manager. *ACM Computing Surveys (CSUR)* (1981).

[35] Gray, J., and Reuter, A. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[36] Gray, J. N. Notes on data base operating systems. In *Operating Systems*. Springer, 1978.

[37] Haerder, T., and Reuter, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. 15*, 4 (1983).

[38] Hagmann, R. Reimplementing the Cedar file system using logging and group commit. In *Proc. of SOSP* (1987).

[39] Haria, S., Hill, M. D., and Swift, M. M. MOD: Minimally ordered durable datastructures for persistent memory. In *Proc. of ASPLOS* (2020).

[40] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[41] Hsu, T. C.-H., Brügner, H., Roy, I., Keeton, K., and Eugster, P. NVthreads: Practical persistence for multi-threaded applications. In *Proc. of EuroSys* (2017).

[42] Hu, Q., Ren, J., Badam, A., Shu, J., and Moscibroda, T. Log-structured non-volatile main memory. In *Proc. of USENIX ATC* (2017).

[43] Huang, Y., Pavlovic, M., Marathe, V. J., Seltzer, M., Harris, T., and Byan, S. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *Proc. of USENIX ATC* (2018).

[44] Hwang, D., Kim, W.-H., Won, Y., and Nam, B. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *Proc. of FAST* (2018).

[45] Izraelevitz, J., Kelly, T., and Kolli, A. Failure-atomic persistent memory updates via JUSTDO logging. In *Proc. of ASPLOS* (2016).

[46] Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y. J., Wang, Z., Xu, Y., Dulloor, S. R., Zhao, J., and Swanson, S. Basic performance measurements of the Intel Optane DC Persistent Memory Module. *CoRR abs/1903.05714* (2019). http://arxiv.org/abs/1903.05714.

[47] jemalloc. http://jemalloc.net/.

[48] Kadekodi, R., Lee, S. K., Kashyap, S., Kim, T., Kolli, A., and Chidambaram, V. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proc. of SOSP* (2019).

[49] Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S. H., and Choi, Y.-R. SLM-DB: Single-level key-value store with persistent memory. In *Proc. of FAST* (2019).

[50] Kannan, S., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Wang, Y., Xu, J., and Palani, G. Designing a true direct-access file system with DevFS. In *Proc. of FAST* (2018).

[51] Kannan, S., Bhat, N., Gavrilovska, A., Arpaci-Dusseau, A., and Arpaci-Dusseau, R. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proc. of USENIX ATC* (2018).

[52] Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. All about Eve: Execute-verify replication for multi-core servers. In *Proc. of OSDI* (2012).

[53] Kim, W.-H., Seo, J., Kim, J., and Nam, B. ClfB-Tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Trans. Storage 14*, 1 (Feb. 2018).

[54] Kumar, V., and Hsu, M. *Recovery mechanisms in database systems*. Prentice Hall PTR, 1997.

[55] Kwon, Y., Fingler, H., Hunt, T., Peter, S., Witchel, E., and Anderson, T. Strata: A cross media file system. In *Proc. of SOSP* (2017).

[56] Lee, S. K., Lim, K. H., Song, H., Nam, B., and Noh, S. H. WORT: Write optimal radix tree for persistent memory storage systems. In *Proc. of FAST* (2017).

[57] Lee, S. K., Mohan, J., Kashyap, S., Kim, T., and Chidambaram, V. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proc. of SOSP* (2019).

[58] Li, J., Michael, E., Sharma, N. K., Szekeres, A., and Ports, D. R. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proc. of OSDI* (2016).

[59] Linux Kernel Organization. Direct access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[60] Liu, Q., Izraelevitz, J., Lee, S. K., Scott, M. L., Noh, S. H., and Jung, C. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *Proc. of MICRO* (2018).

[61] Liu, S., Seemakhupt, K., Wei, Y., Wenisch, T., Kolli, A., and Khan, S. Cross-failure bug detection in persistent memory programs. In *Proc. of ASPLOS* (2020).

[62] Liu, S., Wei, Y., Zhao, J., Kolli, A., and Khan, S. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proc. of ASPLOS* (2019).

[63] Lu, H. J., Matz, M., Girkar, M., Hubička, J., Jaeger, A., and Mitchell, M. System V application binary interface AMD64 architecture processor supplement (with LP64 and ILP32 programming models) version 1.0, 2018. https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf.

[64] Lu, Y., Shu, J., Chen, Y., and Li, T. Octopus: An RDMA-enabled distributed persistent memory file system. In *Proc. of USENIX ATC* (2017).

[65] Marathe, V. J., Mishra, A., Trivedi, A., Huang, Y., Zaghloul, F., Kashyap, S., Seltzer, M., Harris, T., Byan, S., Bridge, B., and Dice, D. Persistent memory transactions. *CoRR abs/1804.00701* (2018). http://arxiv.org/abs/1804.00701.

[66] Marathe, V. J., Seltzer, M., Byan, S., and Harris, T. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *Proc. of HotStorage* (2017).

[67] Memaripour, A., Badam, A., Phanishayee, A., Zhou, Y., Alagappan, R., Strauss, K., and Swanson, S. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *Proc. of EuroSys* (2017).

[68] Memaripour, A., Izraelevitz, J., and Swanson, S. Pronto: Easy and fast persistence for volatile data structures. In *Proc. of ASPLOS* (2020).

[69] Michael, E., Ports, D. R., Sharma, N. K., and Szekeres, A. Recovering shared objects without stable storage. In *Proc. of DISC* (2017).

[70] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* (1992).

[71] Mohan, J., Martinez, A., Ponnapalli, S., Raju, P., and Chidambaram, V. Finding crash-consistency bugs with bounded black-box crash testing. In *Proc. of OSDI* (2018).

[72] Moraru, I., Andersen, D. G., and Kaminsky, M. There is more consensus in egalitarian parliaments. In *Proc. of SOSP* (2013).

[73] Nalli, S., Haria, S., Hill, M. D., Swift, M. M., Volos, H., and Keeton, K. An analysis of persistent memory use with WHISPER. In *Proc. of ASPLOS* (2017).

[74] Nam, M., Cha, H., Choi, Y.-R., Noh, S. H., and Nam, B. Write-optimized dynamic hashing for persistent memory. In *Proc. of FAST* (2019).

[75] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. Scaling memcache at Facebook. In *Proc. of NSDI* (2013).

[76] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast crash recovery in RAMCloud. In *Proc. of SOSP* (2011).

[77] Oukid, I., Lasperas, J., Nica, A., Willhalm, T., and Lehner, W. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proc. of SIGMOD* (2016).

[78] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *Proc. of NSDI* (2019).

[79] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proc. of OSDI* (2014).

[80] Persistent memory development kit. https://pmem.io/pmdk/.

[81] Persistent memory development kit—the libpmemobj library. https://pmem.io/pmdk/libpmemobj/.

[82] pmem/pmem-redis: A version of redis that uses persistent memory. https://github.com/pmem/pmem-redis.

[83] Protocol Buffers | Google Developers. https://developers.google.com/protocol-buffers/.

[84] RAAD, A., WICKERSON, J., NEIGER, G., AND VAFEIADIS, V. Persistency semantics of the Intel-X86 architecture. *Proc. ACM Program. Lang. 4*, POPL (Dec. 2019).

[85] Redis persistence. https://redis.io/topics/persistence.

[86] redis/README.md at 4.0 • redis/redis. https://github.com/redis/redis/blob/4.0/README.md.

[87] SCHWALB, D., DRESELER, M., UFLACKER, M., AND PLATTNER, H. NVC-Hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proc. of IMDM* (2015).

[88] SHULL, T., HUANG, J., AND TORRELLAS, J. AutoPersist: An easy-to-use Java NVM framework based on reachability. In *Proc. of PLDI* (2019).

[89] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proc. of ATEC* (1996).

[90] TWEEDIE, S. EXT3, journaling filesystem, 2000. http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html.

[91] TWEEDIE, S. C., ET AL. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo* (1998).

[92] VAN RENEN, A., VOGEL, L., LEIS, V., NEUMANN, T., AND KEMPER, A. Persistent memory I/O primitives. In *Proc. of DaMoN* (2019).

[93] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of FAST* (2011).

[94] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. of EuroSys* (2014).

[95] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proc. of ASPLOS* (2011).

[96] WANG, C., YING, V., AND WU, Y. Supporting legacy binary code in a software transaction compiler with dynamic binary translation and optimization. In *Proc. of CC* (2008).

[97] WANG, T., LEVANDOSKI, J., AND LARSON, P. Easy lock-free indexing in non-volatile memory. In *Proc. of ICDE* (2018).

[98] swapnilh/whisper: WHISPER is a comprehensive benchmark suite for emerging persistent memory technologies. https://github.com/swapnilh/whisper.

[99] WU, M., ZHAO, Z., LI, H., LI, H., CHEN, H., ZANG, B., AND GUAN, H. Espresso: Brewing Java for more non-volatility with non-volatile memory. In *Proc. of ASPLOS* (2018).

[100] WU, X., NI, F., ZHANG, L., WANG, Y., REN, Y., HACK, M., SHAO, Z., AND JIANG, S. NVMcached: An NVM-based key-value cache. In *Proc. of APSys* (2016).

[101] WU, X., QIU, S., AND NARASIMHA REDDY, A. L. SCMFS: A file system for storage class memory and its extensions. *ACM Trans. Storage 9*, 3 (Aug. 2013).

[102] WU, Z., LU, K., NISBET, A., ZHANG, W., AND LUJÁN, M. PMThreads: Persistent memory threads harnessing versioned shadow copies. In *Proc. of PLDI* (2020).

[103] XIA, F., JIANG, D., XIONG, J., AND SUN, N. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proc. of USENIX ATC* (2017).

[104] XU, J., KIM, J., MEMARIPOUR, A., AND SWANSON, S. Finding and fixing performance pathologies in persistent memory software stacks. In *Proc. of ASPLOS* (2019).

[105] XU, J., AND SWANSON, S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. of FAST* (2016).

[106] XU, J., ZHANG, L., MEMARIPOUR, A., GANGADHARAIAH, A., BORASE, A., DA SILVA, T. B., SWANSON, S., AND RUDOFF, A. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proc. of SOSP* (2017).

[107] YANG, J., IZRAELEVITZ, J., AND SWANSON, S. Orion: A distributed file system for non-volatile main memories and RDMA-capable networks. In *Proc. of FAST* (2019).

[108] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *Proc. of FAST* (2020).

[109] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proc. of FAST* (2015).

[110] Yao, T., Zhang, Y., Wan, J., Cui, Q., Tang, L., Jiang, H., Xie, C., and He, X. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *Proc. of USENIX ATC* (2020).

[111] brianfrankcooper/ycsb: Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[112] Zhang, I., Liu, J., Austin, A., Roberts, M. L., and Badam, A. I'm not dead yet! The role of the operating system in a kernel-bypass era. In *Proc. of HotOS* (2019).

[113] Zhang, I., Sharma, N. K., Szekeres, A., Krishnamurhty, A., and Ports, D. R. K. Building consistent transactions with inconsistent replication. In *Proc. of SOSP* (2015).

[114] Zhang, L., and Swanson, S. Pangolin: A fault-tolerant persistent memory programming library. In *Proc. of USENIX ATC* (2019).

[115] Zheng, S., Hoseinzadeh, M., and Swanson, S. Ziggurat: A tiered file system for non-volatile main memories and disks. In *Proc. of FAST* (2019).

[116] Zuo, P., and Hua, Y. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems 29*, 5 (2018).

[117] Zuo, P., Hua, Y., and Wu, J. Write-optimized and high-performance hashing index scheme for persistent memory. In *Proc. of OSDI* (2018).

[118] Zuriel, Y., Friedman, M., Sheffi, G., Cohen, N., and Petrank, E. Efficient lock-free durable sets. *Proc. ACM Program. Lang. 3*, OOPSLA (Oct. 2019).