2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)

NVOverlay: Enabling Efficient and Scalable High-Frequency Snapshotting to NVM

Ziqi Wang
Carnegie Mellon University
ziqiw@cs.cmu.edu

Chul-Hwan Choo
Samsung Electronics
chulhwan.choo@samsung.com

Michael A. Kozuch
Intel Labs
michael.a.kozuch@intel.com

Todd C. Mowry
Carnegie Mellon University
tcm@cs.cmu.edu

Gennady Pekhimenko
University of Toronto
pekhimenko@cs.toronto.edu

Vivek Seshadri
Microsoft Research India
visesha@microsoft.com

Dimitrios Skarlatos
Carnegie Mellon University
dskarlat@cs.cmu.edu

***Abstract*—The ability to capture frequent (per millisecond) persistent snapshots to NVM would enable a number of compelling use cases. Unfortunately, existing NVM snapshotting techniques suffer from a combination of persistence barrier stalls, write amplification to NVM, and/or lack of scalability beyond a single socket. In this paper, we present *NVOverlay*, which is a scalable and efficient technique for capturing frequent persistent snapshots to NVM such that they can be randomly accessed later. NVOverlay uses *Coherent Snapshot Tracking* to efficiently track changes to memory (since the previous snapshot) across multi-socket parallel systems, and it uses *Multi-snapshot NVM Mapping* to store these snapshots to NVM while avoiding excessive write amplification. Our experiments demonstrate that NVOverlay successfully hides the overhead of capturing these snapshots while reducing write amplification by 29%–47% compared with state-of-the-art logging-based snapshotting techniques.**

***Index Terms*—Non-Volatile Memory (NVM); Snapshotting; Shadow Paging**

I. Introduction

While Byte-Addressable Non-Volatile Memory (NVM) technology[1] has led to many exciting ideas on how to leverage persistence [2, 3, 5, 8, 11, 23, 31–33, 35, 40, 41, 46, 47, 55, 57, 62, 68, 69, 76, 78, 84–87, 90–92], our focus in this paper is specifically on the benefits and challenges of using NVM to support *frequent persistent snapshotting*. In particular, our goal is to capture persistent snapshots of the *full physical address space* of a process to NVM at a rate of hundreds of times per second (i.e. on the order of milliseconds).

Usage models enabled by frequent persistent snapshots. The ability to efficiently capture frequent snapshots to NVM would enable a form of *persistent, multiversioned* memory system, which in turn enables the following four usage models: (1) *time-traveling* or *record-and-replay* debugging [52, 77] of distributed cloud applications, where snapshots are captured upon user-specified events (aka "watch points") that often happen in a bursty fashion (as computations of interest flow across the distributed system); (2) implementing persistent and durable data structures [15]; (3) fine-grained system backup and replication [88]; and (4) low-latency crash recovery.

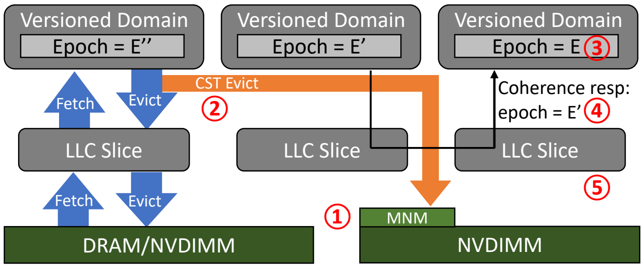


Fig. 1: **NVOverlay Features** – ① – Multi-snapshot NVM Mapping (MNM); ② – Coherent Snapshot Tracking (CST) and background persistence; ③ – Distributed epochs; ④ – Coherence-driven epoch synchronization; ⑤ – Distributed LLC support.

Key challenges: Tracking changes to memory and avoiding NVM write amplification. There are two fundamental challenges in supporting high-frequency persistent snapshotting so that it is both efficient and scalable. The first challenge arises from the fact that when it is time to create a snapshot, we want to quickly gather the minimal amount of data that must be stored in the snapshot. Hence the *first* challenge is *keeping track of exactly what changes have occurred* within the physical address space since the previous snapshot. Keeping track of these "deltas" is difficult both because we want to capture them without slowing down normal execution, and because we want to capture them *coherently* across entire parallel applications running on scalable shared-address space systems (spanning multiple sockets). Once we have successfully identified the information in memory that should be part of a snapshot, the *second* fundamental challenge is making that snapshot *persistent* by efficiently writing it to NVM so that it can be randomly-accessed later (as one of multiple snapshots) while *avoiding NVM write amplification*.

Previous work. There have been a number of software [7, 19, 21, 30, 37, 44] and hardware [14, 20, 22, 27–29, 36, 53, 59–61, 65, 73, 82, 89] proposals for capturing persistent snapshots to NVM. Unfortunately these previous designs suffer from a combination of non-trivial performance overheads (due to persistence barrier stalls or NVM write amplification), lack

[*] This work is supported in part by grants from NSF, Samsung, and Intel.

[1] In this paper, we focus on NVM devices in DIMM form factors, such as Intel® Optane^{TM} persistent memory.

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00046

of scalability to multi-socket systems, and/or lack of support for randomly accessing multiple previous snapshots (which is important for the distributed debugging usage model).

Software-based approaches generally suffer performance degradation due to persistence barriers [30, 37]. In addition, the software logging approaches cause NVM write amplification by first writing the snapshot data to a persistent log before later copying it to its eventual location on the NVM (where it can be randomly-accessed by address). This write amplification effectively halves the NVM bandwidth, as well as reducing the number of Program/Erase (P/E) cycles.

Hardware proposals improve performance by overlapping normal execution with making a snapshot persistent via special hardware that controls the write back of dirty data in the caches to NVM [14, 20, 22, 27–29, 36, 53, 59–61, 65, 73, 82, 89]. While these proposals effectively eliminate persistence barrier stalls, they have two major drawbacks. First, they typically suffer NVM write amplification due to logging (as discussed above). Second, they typically do not scale to multi-socket systems, making assumptions such as inclusive monolithic cache hierarchies within a single socket, and/or introducing non-scalable structures such as centralized mapping structures [65] or control logic [59].

Key insights in our approach: NVOverlay. To overcome both of the key challenges in supporting frequent persistent snapshots, our NVOverlay design builds upon two key insights. First, to address the challenge of tracking the changes to memory since the most recent snapshot in a way that is both efficient and scalable, we combine a novel extension of *multiversioned page overlays* [71, 79] with a set of *relaxed distributed epochs* (maintained using a form of Lamport clock [16, 38]) to create our new **Coherent Snapshot Tracking (CST)** mechanism. Second, to address the challenge of avoiding write amplification while persisting a set of multiple snapshots to NVM, we avoid logging altogether, and instead use a form of *persistent shadow-mapping at the NVM interface* in our **Multi-snapshot NVM Mapping (MNM)** mechanism. Details of both of these mechanisms are discussed later in Sections IV and V, respectively, and are illustrated in Figure 1.

Our paper makes the following contributions:

- We propose and evaluate NVOverlay, which supports efficient and scalable high-frequency snapshotting to NVM, thereby enabling a number of usage models (including distributed debugging);
- NVOverlay uses Coherent Snapshot Tracking (CST) to efficiently track changes to memory since the last snapshot, scaling to multi-socket systems;
- NVOverlay uses Multi-snapshot NVM Mapping (MNM) to support persistence of multiple snapshots to NVM while avoiding write amplification;
- Our performance evaluation of NVOverlay demonstrates that it can successfully hide most of the overhead of creating frequent persistent snapshots, while reducing NVM write amplification by 29%–47%.

II. Motivation and Related Work

A. Failure Atomicity with Persistence Barriers

To achieve failure atomicity with NVM, today's application programmers have been using various software solutions (since proposed hardware solutions are not yet available). In particular, these software-based approaches typically include transactional libraries [7, 10, 19, 21, 22, 24, 44, 45, 54, 61], ad-hoc data structures [2, 8, 11, 23, 40, 41, 57, 62, 69, 76, 78, 86, 91], memory allocators [5, 55, 68], and/or storage services [3, 31, 35, 84, 85]. All of these software approaches rely on *persistence barriers* [30, 37] to enforce write ordering, which is critical for atomicity. For example, with undo logging, the log entry is forced to be flushed to NVM before data is updated. In Romulus [10] (which uses software shadow mapping), the next transaction can only start after the working set of the prior one becomes persistent.

A persistence barrier typically consists of a series of cache line flush instructions (e.g., `clwb`/`clflush`) followed by a memory fence (e.g., `sfence`). Frequent usage of persistence barriers negatively impacts performance, however, because (1) the pipeline stalls while the flushes are processed and (2) the execution of multiple barriers may be serialized unnecessarily.

B. Overlapping Persistence with Execution

To eliminate software persistent barriers, previous work has proposed adding special-purpose hardware to enforce persistence in the background. For example, *hardware logging* [14, 20, 27–29, 36, 59, 73, 81, 89] generates log entries and coordinates data write-backs in the background, thereby overlapping persistence with execution. These persistence logs can be managed by the load/store unit [73], the coherence controller [36], the cache controller [14, 27, 28, 59, 81, 89], or the memory controller [20, 29].

While these hardware-based logging approaches have minimal runtime overhead compared with software approaches, their main disadvantage is *NVM write amplification*. Because these logging approaches write both the log data and the dirty data back to the NVM, they typically incur a write amplification factor of at least two (or three in the case of undo+redo logging [54, 61]). Excessive write backs from the cache hierarchy to NVM can degrade system performance by wasting bus bandwidth, and they can also reduce the lifespan of NVM devices (given their limited number of writes before wear out [17]).

C. Reducing Write Amplification via Shadowing

To avoid the NVM write amplification of hardware logging, an alternative approach is *hardware shadow paging* [60, 65, 82], which remaps dirty data to an alternate "shadow address" to avoid overwriting the current consistent image. Because shadow paging only writes to data once, it has no inherent data write amplification. The remapping can be performed by the TLB [60] or the memory controller [65, 82]. Hardware shadow paging can also perform persistence operations in the background, to help minimize runtime overheads.

Features Designs	Minimum Write Amplification	No Commit Time Flush	No Read Redirection	Software Persistence Barrier	Unbounded Working Set	Supports Non-Inclusive LLC	Distributed Versioning Protocol
SW Undo Logging	×	✓	✓	Per Write	✓	✓	×
SW Redo Logging	×	×	×	Constant	✓	✓	×
SW Shadow Paging	Maybe	×	×	Constant	✓	✓	×
PiCL [59] (HW Logging)	×	✓	✓	None	✓	×	×
SSP [60] (HW Shadow)	✓	×	×	None	×	✓	×
NVOverlay	✓	✓	✓	None	✓	✓	✓

TABLE I: **Qualitative Comparison of NVOverlay with Other Designs**

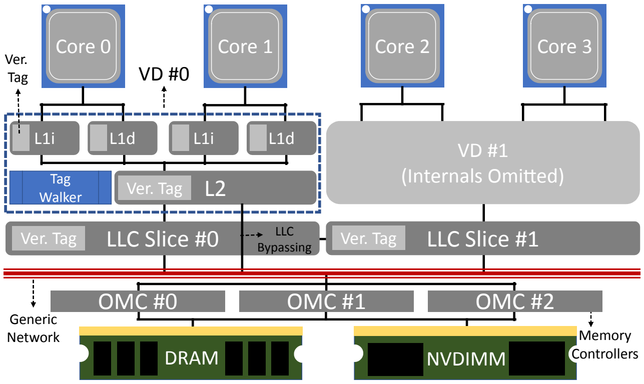


Fig. 2: **System Architecture** – Components above the system memory bus (the red bar) constitute the CST frontend; Components below the memory bus constitute the MNM backend.

While we believe that hardware shadow paging is generally on the right track, existing designs face several challenges: e.g., restrictions on working set sizes, support for only a limited number of snapshots, and non-overlappable mapping table updates. NVOverlay is a novel approach to a fine-grained shadowing model that avoids these drawbacks, as we will discuss in detail in Section II-E.

D. Scalability

Prior persistent snapshotting proposals have suffered from a number of scalability challenges. First, because modern multicore processors feature non-inclusive, distributed LLC slices [4, 26, 34, 56, 80], proposals with a centralized LLC tag walker [59] or control logic [89] simply will not work. Second, most previous proposals assumed a globally synchronized epoch [59, 65, 82]; achieving this consensus is difficult to scale due to increasing communication costs. Finally, previous designs tend to generate bursts of traffic (especially when snapshots occur frequently), due to coordinated, simultaneous write backs from all components. As systems scale, these bursts of traffic become increasingly likely to hurt performance by saturating the bandwidth of memory buses and NVM devices.

E. Our Approach: NVOverlay

At a high level, NVOverlay adopts a combination of barrier-less Coherent Snapshot Tracking (CST) frontend with overlapped persistence and find-grained, shadow-mapped Multi-snapshot NVM Mapping (MNM) backend. This combination eliminates both unnecessary barrier stalls on the frontend, and log write amplification on the backend.

NVOverlay assumes an epoch-based snapshot model, where the execution is divided into disjoint intervals, called "epochs", which is the basic unit of snapshotting. NVOverlay maintains working data and snapshot data separate. Working data is maintained in the ordinary manner in either NVM or DRAM. Dirty data generated in different epochs will be persisted to the NVM as separate copies, which can be accessed independently.

Both the frontend and the backend are designed with scalability as one of the major design goals. To eliminate centralized epochs and control logic, we relax the consistency requirements of snapshots by allowing them to be taken in a state not necessarily conforming to any real-time state that has occurred in the system, but yet still consistent in terms of causality, which is defined by the coherence protocol (see Section III-C). To achieve this, we partition the cache hierarchy above the LLC into Versioned Domains (VDs), and let each of them maintain its own epoch. All epoch counters in the system form a Lamport clock [38], which are also updated in a similar way: a local epoch counter j is updated to a remote epoch counter i if and only if the local epoch observes a dirty cache line generated by the remote epoch and $i > j$.

Dirty data generated by VDs are tracked collectively by the version-tagged hierarchy, where every logical cache line is tagged by an extra version field indicating the epoch in which the line is last written. NVOverlay mandates the invariant that dirty lines of version E become immutable after epoch E ceases to be active. This way, multiple instances of the same address co-exist in the hierarchy, each constituting part of different snapshots. The Version Coherence Protocol, a simple addition to the existing coherence protocol, tracks in-cache versions and orchestrates the eviction to ensure correct ordering. Meanwhile, the backend, on receiving a dirty line of version E, inserts it into a per-epoch mapping table which enables random, cache line granularity accesses. These tables are also continuously merged into a persistent global Master Table that maps the current consistent memory image. Data movement is not needed during the merge, as only table entries are copied.

A qualitative comparison between NVOverlay and other similar designs are presented in Table I.

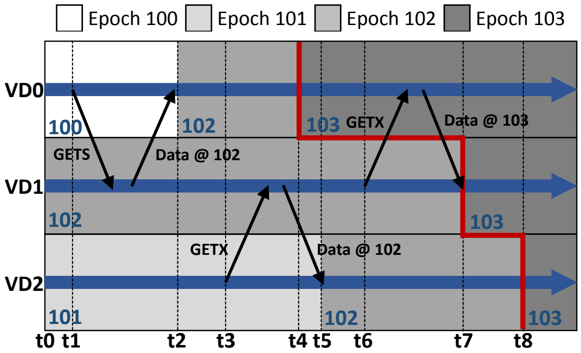


Fig. 3: **Relaxed Epoch Model** – The red line shows the system state captured at epoch 102 (t4, t7, t8 for VD0, VD1 and VD2, respectively).

III. NVOverlay Insights

A. Page Overlays

Page Overlays [71, 79] was originally proposed as a fine-grained address mapping scheme that allows a virtual address to be mapped to multiple backing store addresses at cache line granularity. Each cache line in the hierarchy is tagged with a **Overlay ID (OID)**. A single address tagged with different OIDs could be mapped to different physical locations by the Overlay Memory Controller (OMC), which serves as the memory controller sitting between the cache hierarchy and the main memory.

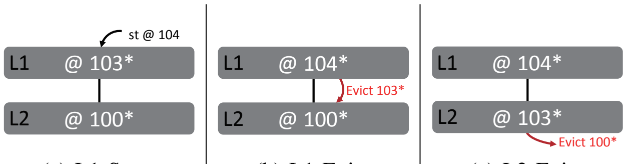
Readers are not required to possess prior knowledge of Page Overlays in order to understand NVOverlay. Our work is mostly self-contained, while remaining compatible with the original design. Readers interested in the original Page Overlays design and a comparison with NVOverlay are encouraged to read the paper [71] for more info.

B. Architecture

Fig. 2 depicts the system architecture. We assume a multicore system with a distributed last-level cache (LLC). The LLC does not need to be inclusive, as is the case for some large systems [80]. The inclusive L2 cache can be shared by a small number of cores [50]. All cache tags in the hierarchy are extended with a 16 bit *OID* field, which stores the epoch ID in which the line is last updated. In the figure, core 0, core 1, and the shared L2 form a Versioned Domain, *VD0*, while the remaining two cores and the L2 form *VD1*. Although only four cores and two LLC slices are shown, the actual system can be much larger, or even distributed.

For design simplicity, L1 and L2 caches in a VD run the same epoch. Cache controllers maintain their own *cur-epoch* registers for tracking the VD's current epoch, which are synchronized within a VD. Since VDs are relatively small, epoch synchronization is a lightweight event that only incurs local communication.

Snapshot cache lines evicted from the cache hierarchy (not only LLC) are handled by the OMC, which is integrated into the memory controller. The OMC maintains a series of mapping tables, which translates the cache line address to the shadow address on the NVM. As shown by the picture, NVOverlay's control logic can be distributed over multiple OMCs for better scalability, each responsible for its own address partition. Note that the application can use DRAM, or NVM, or both as working memory.



(a) L1 Store (b) L1 Evict (c) L2 Evict

Fig. 4: **L1 Store-Eviction** – Only showing line OID. "*" means dirty version. Additions to baseline protocol are marked red.

C. The Relaxed Epoch Model

Conceptually, NVOverlay divides the execution of a single VD into **epochs**, identified by 16-bit integers. For simplicity of discussion, we first assume only a single VD is present. Within each epoch, the system state is updated by store instructions.[2] Such state changes are incrementally captured by NVOverlay's **Coherent Snapshot Tracking** (see Section IV), and persisted to the NVM as a **snapshot**. Each snapshot only contains state changes made within that epoch, but not before. Processors in the VD also dump their internal context to the NVM at the end of every epoch as part of the snapshot.

On crash recovery, NVOverlay first searches the most recent fully persisted epoch, E. Then the consistent memory image is rebuilt by combining all incremental changes before and during E, taking time proportional to the working set size.

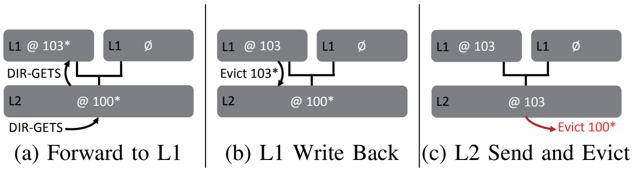
The model becomes more complicated, due to data dependencies, when multiple VDs interact via shared memory accesses. Recall that, unlike previous proposals, each VD in NVOverlay runs an independent epoch. How could data dependency be observed, for example, if a cache line written by VD X in epoch i is accessed by VD Y in epoch j?

As a solution, NVOverlay synchronizes VD epochs when data "from the future" is observed, setting VD Y's local epoch to i if $j < i$ in the above example. This is similar to how a Lamport clock captures ordering of events in a distributed system [16, 38]. As a trade-off, the "relaxed" snapshot taken by NVOverlay may not be the exact memory image at any real-time point during execution. Nevertheless, the snapshot still correctly preserves system progress, since the image after recovery is consistent in terms of logical time. An example is given in Fig. 3. In this figure, the captured snapshot only reflects real-time memory states of *VD0*, *VD1*, and *VD2* in time *t5*, *t7* and *t8*, respectively. The snapshot is still consistent, however, since the local VD state it captures agrees with the causality order implied by inter-VD cache coherence.

IV. Coherent Snapshot Tracking (CST)

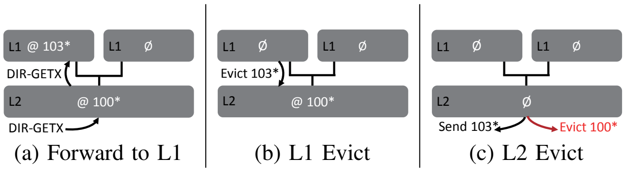
As described in Section III-C, the Coherent Snapshot Tracking (CST) design must address two challenges: incremental

[2] This paper primarily focuses on recording memory state changes; for complete recovery, certain I/O and external events (network data from a *recv* syscall) may also need to be recorded at the system level. See, e.g., [49, 74, 77].



(a) Forward to L1 (b) L1 Write Back (c) L2 Send and Evict

Fig. 5: **L2 External Downgrade** – After downgrade, both caches hold a shared copy of the most recent version.



(a) Forward to L1 (b) L1 Evict (c) L2 Evict

Fig. 6: **L2 External Invalidation** – Optimization is applied. Version 103 is directly sent to the requestor via cache-to-cache transfer.

tracking of state changes and synchronization of epochs. In the following sections we discuss these two topics in details. We begin with operations within a single VD, and then extend the discussion to multiple VDs and with cache coherence.

We assume directory-based MESI [63] as the baseline protocol. The design can be easily extended to support snoop-based MESI, or its mainstream derivations such as MOESI [12] or MESIF [75]. We also emphasize that NVOverlay does not modify the baseline protocol. Instead, only a few extra tag checks and evicts are added to existing coherence actions. States and transitions remain untouched.

A. *Version Access Protocol*

Versions are cache lines whose contents are produced during epoch execution. A cache line's version number is the value of its OID tag, which is set to the VD's epoch number when the line is written. All coherence messages sent over the network also contain a version number, *RV (Request/Response Version)*, the meaning of which will be explained below.

Versions can be either clean or dirty depending on its coherence state. For example, in MESI protocol, `M` state lines are dirty, while `S` and `E` state are clean. NVOverlay maintains the invariant that clean versions are already persisted on NVM. A dirty version from a previous epoch *E'* is, therefore, immutable in epoch *E*, since it might be part of the snapshot state of *E'* that has not been persisted.

The goal of the version access protocol is to ensure that only the most up-to-date version is accessed, even when multiple versions of the same address may co-exist in the hierarchy. The protocol also guarantees that eventually all versions in the hierarchy are evicted to NVM, while DRAM only keeps the most recent version as the working copy.

We next describe the version access protocol in L1 and L2 caches respectively.

1) L1 Operations: **On receiving a load request** from the processor, the L1 behaves exactly the same as in a non-overlay system. In particular, the tag lookup is performed without checking the OID tag, as opposed to original Page Overlay's lookup protocol where both address and OID are compared. If the lookup misses, a `GETS` request is sent to L2 before the load is retried. Otherwise, the load completes locally.

On receiving a store request, a tag lookup is performed as in loads. The store request's *RV* is set to the VD's *cur-epoch*. If the lookup signals a miss, or if the line is not in writable state (`E` or `M`), the cache controller will first acquire exclusive permission by sending `GETX` before retrying the store. The controller then compares line OID with *RV*. If the line is dirty and OID equals *RV*, the store completes locally. Otherwise, the version is immutable. In this case, the L1 controller evicts the immutable version to L2 without invalidating the line, and performs the store in-place after the eviction is scheduled (see Fig. 4). Such "store-eviction" is critical in NVOverlay's design, since it enables caching multiple versions in the hierarchy, while leveraging the inclusive L2 cache as a temporary buffer for older versions. The line OID is also updated to *RV* to reflect the fact that the line is now in epoch *RV*'s snapshot.

On line eviction, if the line is dirty, a `PUTX` request is scheduled in L1's evict buffer, with *RV* set to line OID. Whether clean evictions are processed is implementation-dependent. Since cache line evictions are not on the critical path, store-eviction will not affect L1 cache access latency.

2) L2 Operations: **On receiving a `GETS` or `GETX`** from L1, the L2 performs a tag lookup as usual, and signals a miss if the block is not present or has insufficient permission. When the requested block is in the L2, it is read out and sent to the L1 as response, the *RV* of which is set to line OID.

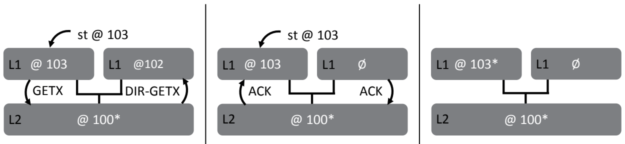
On receiving a `PUTX` from L1, the L2 first performs a tag lookup to read out the line OID and coherence state. If the line is dirty, and $OID < RV$, then L2 evicts the current line to avoid overwriting an old version. This also preserves the invariant that L1 versions must be no smaller than the L2 version on the same address. Evictions scheduled in this situation will not invalidate the L1 copy, since inclusiveness still holds. In all cases, the L2 completes the request by copying data and OID into the cache slot.

On line eviction, if the line is a dirty version, in addition to sending it to LLC, the L2 controller also sends the version to OMC via the coherence network, bypassing the LLC. Note that in a non-inclusive LLC design, such bypassing network already exists to allow the L2 cache directly writing back lines that are predicted "dead" [18]. Our design, therefore, avoids adding a dedicated datapath, and just takes advantage of LLC bypassing. The request's *RV* is also set to the line OID. Backend operations are discussed in Section V.

3) External Invalidation and Downgrade: The main challenge of implementing external invalidation and downgrade is the fact that L1 and L2 may each cache a dirty version. NVOverlay solves this with extra evictions, as shown below.

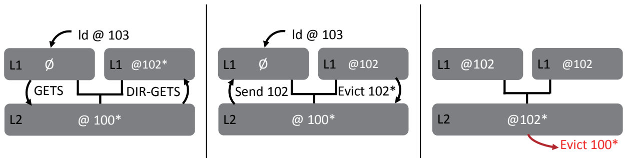
On receiving an external invalidation (`DIR-GETX`) or downgrade (`DIR-GETS`), the L2 controller first queries its own directory for any L1 sharer. If there is none, L2 handles the request locally by scheduling an eviction for the requested version. The line state is also set to `I` or `S` respectively.

If there are L1 sharers, the L2 controller first forwards the request to them. The L1 controller simply schedules an eviction,



(a) L2 Forward Req (b) Inv. ACK (c) Store Completes

Fig. 7: **Intra-VD Invalidation** – Assume directory forwards ACK.



(a) L2 Forward Req (b) Forward Data (c) L2 Evict

Fig. 8: **Intra-VD Downgrade** – Assume directory forwards data.

before changing the line state. The L2 controller then processes the eviction from L1, if any, and handles the request locally. In both cases, the most up-to-date version is also sent back to the directory as response, with response's *RV* setting to line OID. An example is given in Fig. 5.

One difference between our protocol and the baseline is that two evictions, instead of one, may be generated during the process, which doubles the eviction traffic on *both* the LLC and OMC, as every version written back from the L2 to the LLC to fulfill forwarded coherence requests needs to be sent to the OMC. This happens when L1 and L2 both have cached dirty versions, and their OIDs are different, as shown by Fig. 6. A closer inspection reveals, however, that the excessive evictions can be avoided by leveraging two simple observations. First, if both L1 and L2 have dirty versions, the L2 version need not be evicted to LLC, since it does not constitute the current memory image (the L1 version is newer). Second, if the request is an invalidation, and the current VD owns the most recent version, then this version need not be sent to the LLC directory (and hence also sent to the OMC) to fulfill the `DIR-GETX` request at all. Instead, a cache-to-cache transfer is initiated to send the line directly to the requestor cache[3], reducing both write back traffic and coherence latency (see Fig. 6).

4) LLC and DRAM Operations: Once a version leaves a VD, it is guaranteed to be persisted, even if the coherence state may still indicate dirty. The LLC and DRAM, therefore, do not implement the version coherence protocol, except that line OIDs are updated on write backs.

To maintain per-line OID in the DRAM, the DRAM controller may reserve a few words for every DRAM page, and updates both OID and data in a way similar to how ECC memory is updated. In fact, the 16-bit OID can just be stored in the ECC banks on ECC-enabled memory. Other techniques, such as DRAM compression [72], can also be employed to embed OID without extra cost.

By preserving line OIDs even outside of VDs, we avoid losing track of the most recent epoch that updates the line. This is necessary for "remembering" data dependency.

B. Coherence-Driven Epoch Update

1) Versioned Domain Coherence: Intra-VD coherence remains unchanged, except that when L1 writes back a dirty version, the L2 needs to check its local version and possibly schedules an eviction. Examples are given in Fig. 7 and Fig. 8.

When caches request an address not present or has insufficient permission in the VD, the request becomes inter-VD, which is forwarded by the L2 controller to the LLC directory. The directory further forwards the request (or invalidations, downgrades) to other VDs, or to the LLC, exactly as in a non-NVOverlay cache hierarchy.

2) Advancing Epochs: When an inter-VD request receives the response, the *RV* field of the response is always set to the OID of the line. On receiving such a response, the L2 controller compares its *cur-epoch* register with *RV*. If *RV* is larger, the VD must terminate the current epoch, and advance to epoch *RV*.

In order to advance the epoch, the L2 controller first signals all cores in the VD to stall their pipelines. The L2 cache controller also stops responding to external coherence requests, and drains the intra-VD request queue (deadlocks are impossible). Next, all cores in the VD dump their non-speculative context to the NVM, tagged with *cur-epoch*. Finally, the *cur-epoch* registers in all cache controllers are updated to *RV*.

In practice, to avoid large epoch skews between VDs, VDs also advance their local epochs after a fixed number of instructions or on external events. In the rare event of an epoch wrap-around, please refer to Section IV-D.

C. Cache Tag Walker

The last component of the Coherent Snapshot Tracking mechanism is the L2 cache tag walker, a hardware state machine built into the cache controller. The tag walker runs opportunistically, scanning cache tags only when they are not used by outstanding requests. Each VD has its own tag walker, as shown in Fig. 2.

Dirty versions whose OIDs are smaller than the L2 controller's *cur-epoch* will be written back to the NVM by the tag walker. The write back downgrades the line from `M` state to `E` state, in addition to sending both data and OID to the OMC. The `E` state line will be either discarded when evicted, or overwritten when a dirty line on the same address is evicted from L1. The correctness of NVOverlay protocol does not rely on the tag walker making progress, though.

D. Epoch Wrap-around

Conceptually, epoch numbers should monotonically increase, since they represent the progress of computation. In practice, epoch numbers are represented with a fixed-width integer and will wrap around to zero eventually.

The simplest approach to eliminating errors as the system approaches the wrap-around condition would be to reset system-wide versioning by clearing all local epochs and version tags,

[3] Some coherence protocols have already implemented this trick, i.e., ownership of an address, on dirty invalidations, is not transferred to the LLC, but to the peer cache via point-to-point links. NVOverlay just uses the existing mechanism in this case.

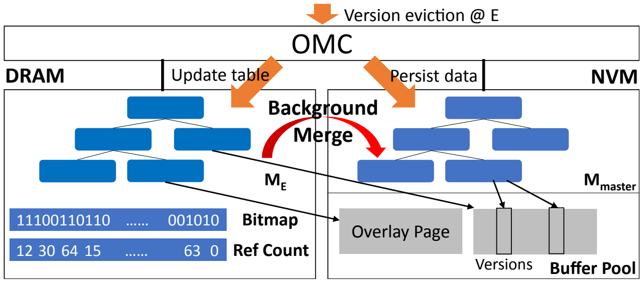


Fig. 9: **Multi-snapshot NVM Mapping** – Orange arrow represents data flow on L2 eviction. Red arrow represents background merge after E becomes recoverable. Blue represents metadata, while grey represents data.

after flushing the cache. The second solution does not force a reset, but limits inter-VD epoch skew to half the version-number space. We partition the epoch space into two equally sized groups, L and U. A persistent *epoch-sense* bit indicates whether epochs in L are larger than those in U, or the other way around (in-group ordering is unchanged). The OMC enforces the invariant that all VDs must be running epochs in the same group. Whenever a VD first advances its local epoch from one group to another, the system ensures that no cache lines remain with tags belonging to the "new" group and flips the *epoch-sense* bit, essentially recycling epoch numbers in the currently smaller group for reuse by "moving" them ahead of the currently larger group.

E. Discussion

Protocol Compatibility: As stated in the beginning, neither does NVOverlay assume specific coherence protocols, nor does it modify the coherence state machine. As long as the protocol supports the notion of "ownership", it can be extended to support NVOverlay.

Coherence Overhead: NVOverlay's version coherence protocol only generates more evictions, which is out of the critical path in most cases. Such eviction overhead also exists in most background persistence designs.

Reducing NVM Writes: If an address is evicted or downgraded by L2 frequently in the same epoch, redundant write backs to NVM (but not to DRAM) will be generated. To this end, we propose adding a battery-backed, write-back cache to OMC to reduce both persist latency and NVM writes. The cache essentially acts as a persistent LLC for absorbing version evictions, which will be flushed on a power failure.

V. Multi-snapshot NVM Mapping (MNM)

A. Overview

Versions evicted from VDs are tracked by the Multi-snapshot NVM Mapping (MNM) mechanism, which can be retrieved on request. NVOverlay's MNM is managed by the OMC in both DRAM and NVM: Data structures that can be rebuilt upon recovery are maintained in volatile DRAM for better access bandwidth and to avoid NVM contention. Meanwhile, versions are maintained in the NVM compactly as overlay data pages. The OMC maintains the image of the most recent recoverable epoch using a series of overlay mapping tables. As we will see below, these tables are continuously updated, in the background, to incorporate more recent versions evicted from the frontend. Fig. 9 depicts the MNM.

B. Determining the Recoverable Epoch

In NVOverlay, an epoch E becomes fully persistent only after all VDs have (1) advanced their local epochs past E; and (2) written back all dirty versions produced in E. In addition, for epoch E to be used for recovery, all epochs before E must have been fully persistent as well. Since epochs are not globally synchronized, the recoverable epoch must be determined in a distributed fashion, as we discuss below.

Each tag walker has a local register, *min-ver*, which is initialized to *cur-epoch* when tag walk begins, and updated to the smallest version OID encountered during the walk. The L2 cache controller then sends the value of *min-ver* to the OMC, where an array of most recently received *min-ver* for each VD in the system is maintained. On receiving the message, the OMC recomputes the recoverable epoch, E_r, as the smallest among all *min-ver*s. E_r is also atomically written to a known location, *rec-epoch*, on the NVM.

If multiple OMCs are present, each OMC first computes its local E_r, and then one of them is selected as the master, to which all remaining OMCs send their E_rs. The master OMC persists the final result after computing the smallest E_r.

C. Overlay Mapping Tables

NVM storage allocated to the multi-snapshot NVM mapping mechanism is maintained as a page buffer pool, which is initialized at system startup time, and managed by OMC hardware. The OMC tracks the allocation status of pages using a bitmap (see Fig. 9) with negligible storage overhead. The index need not be persistent, and can be rebuilt on recovery.

For each epoch E, the OMC maintains a per-epoch volatile mapping table M_E. The mapping table is implemented as a four-level radix tree in DRAM, similar to x86-64 page tables. Table M_E tracks versions produced in epoch E by mapping the physical address of the version to data pages on NVM. The OMC manages epochs as separate page overlay instances, each having one mapping table and a set of distinct data pages. As in the Page Overlays design, sparse pages (pages with only a few versions) are stored compactly in sub-pages smaller than 4KB to save storage (see [71, Sec. 4.4]).

When a version from epoch E is written back, the OMC finds M_{RV} using the request's *RV*, and then inserts the version into M_{RV} using the physical address as key. This process is very similar to how an OS populates its page table, except that NVOverlay uses the 48-bit physical address as table index.

The OMC also maintains a **Master Mapping Table**, M_{master}, which stores overlay mapping for epoch *rec-epoch*. M_{master} is a *five-level* radix tree (one more level than the per-epoch table). The first four levels are exactly the same as per-epoch tables, and the last level is indexed by address bit 6–11 for cache line granularity mapping, as shown in Fig. 10. M_{master} reflects the current consistent memory image, and all nodes of M_{master}

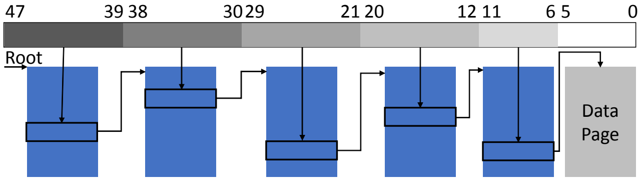


Fig. 10: **Master Mapping Table** – Shows five-level radix tree structure. Blue represents index pages while grey represents data.

are persisted on NVM. Updates to M_{master} are made atomic using either logging or persistent buffers [20]. Whenever *rec-epoch* is updated by the OMC from the old epoch E_i to new epoch E_j ($j > i$), the OMC also merges mappings from volatile tables M_{i+1}, M_{i+2}, ..., M_j, to M_{master}, advancing the consistent memory image. The update procedure simply scans the per-epoch table, and for each version *V*, inserts its physical address and NVM address into M_{master}. No data page is copied in this process. Both table update and merge are performed in the background by the OMC without affecting normal execution.

D. Garbage Collection

Once a table M_E is merged, versions unmapped from M_{master} become stale, which qualify for garbage collection (GC). We cannot, however, immediately reclaim their storage, since versions are mapped in the granularity of pages (or subpages, in the case of sparse page – we use "page" to refer to both below), while unmapped individually in cache line granularity.

Less frequently updated cache lines will block the containing page from being GC'ed, causing storage explosion. When NVM storage runs out, the OMC raises an exception to the OS, which simply allocates more pages, if feasible, and notifies the OMC of the physical address range. Otherwise, if allocating more pages to the OMC is deemed infeasible by the OS, e.g., the device is full, or the storage consumption reaches a quota limit, the OMC needs to perform version compaction to free some space. The algorithm, when invoked, will start from the oldest epoch still having versions mapped by M_{master}, and copy these versions over to the most up-to-date epoch, as if these addresses were written in the current epoch. The source pages can then be safely reclaimed. The resulting write amplification is negligible, as long as the majority of addresses are updated at least once during the past few epochs. DRAM pages used by per-epoch tables can be reclaimed as soon as they are merged into M_{master}.

E. Retrieving Persistent Snapshots

Recall that persistent snapshots are stored on the NVM in a per-epoch manner for efficient retrieval. Depending on the scenario, snapshot data can be accessed either individually using the per-epoch mapping table, or using M_{master} for only accessing the latest snapshot. We next discuss a few use cases where snapshots are utilized.

Crash Recovery: After a crash, the OS or firmware first restores the system to the initial state when snapshotting was started. The recovery procedure then loads the consistent image from the NVM by scanning M_{master} and reading all versions into their corresponding addresses in the DRAM. After this completes, the procedure reads *rec-epoch* stored on NVM, and finds the processor context that were dumped to the NVM at the end of *rec-epoch*. After loading the context, the system resumes execution as if the crash had never happened. Volatile OMC data structures are also rebuilt during the recovery.

Remote Replication: Snapshots, once persisted, can be transferred to a remote backup machine via the network. On receiving the snapshot, the remote machine can either choose to replay the incremental changes as redo logs, or archive them for future accesses. The remote replication architecture and atomicity guarantees, however, are orthogonal to our work, and have been studied quite thoroughly in previous works [88].

Debugging/Time-Travel Reads: For accessing a single address *X* on epoch *E*, the access function finds the largest *E'*, $E' \leq E$, where $M_{E'}$ maps the address *X*. This "fall-through" semantics is similar to how version chains are accessed in an MVCC database [13, 39, 58, 83], which is necessary since snapshots are saved incrementally.

F. Discussion

Hardware Cost: NVOverlay's hardware cost includes per-line 16 bit OID tag field and OMC logic. The 16 bit OID tag only increases total on-chip SRAM storage by at most 3.2%. Similar costs also exist in prior proposals [59, 81, 89]. Neither the replacement algorithm nor the coherence state machine is changed, requiring less verification cost.

The OMC logically serves as the memory controller. It can be either a dedicated CPU thread, as in software managed caches [1], or implemented as hardwired logic [20, 29, 82], or an embedded, programmable micro-controller chip [6, 70].

We emphasize that NVOverlay's hardware additions, such as OID tagged lines, are of general usefulness in many other areas such as transactional memory [79]. Besides, Page Overlay's broad user scenario also enables potentially many novel use cases to be explored once it is deployed.

Runtime DRAM Overhead: Each 64 byte line in the DRAM is tagged with 16 bit OID, incurring a DRAM overhead of at most 3.2%. The OID tracking granularity can also be larger than 64 bytes. For example, the OMC may only reserve 16 OID tags for each 4KB DRAM row, lowering tagging overhead to less than 0.8%. In this case, each OID will be shared by a "super block" of 4 cache lines. The existing OID is only updated if the incoming OID is larger.

Runtime NVM Overhead: NVM pages are used for storing both the working set and M_{master}. Unlike previous shadow paging proposals where a worst-case 2x space overhead is possible [10, 65, 82], NVOverlay allows users to make a trade-off between write amplification and storage by setting a space overhead threshold. If the number of in-use pages exceeds this threshold, OMC will suspend adding new versions, and immediately start version compaction to reclaim storage.

Scaling to Large NVM Arrays: On future NVM platforms, the storage density of which is expected to be higher than DRAM, NVOverlay's OMC-based MNM design is also expected to scale to large NVM arrays. On such systems, multiple memory controllers may co-exist, each responsible for serving requests on an address partition. Each OMC maintains its own instance of overlays and of M_{master} for the address partition. One OMC is selected as the master, which maintains the *min-ver* array for all VDs, which will send their epoch updates to the master.

VI. Experimental Framework

A. Simulation Environment

Our evaluation uses zsim [67], a Pin-based [48] simulator featuring fast, cycle-accurate multicore simulation. We implemented NVOverlay as a separate module without changing the existing coherence protocol. Table II shows the configuration of the simulated system.

Processor	16 cores 4-way superscalar @ 3GHz
L1-D cache	32KB, 64B lines, 8-way, 4 cycles
L2 cache	256KB, 64B lines, 8-way, 8 cycles
Shared LLC	32MB, 64B lines, 16-way, 30 cycles
DRAM	DDR3 1333 MHz, 4 controllers
NVDIMM	16 banks, 133 ns write latency (miss)

TABLE II: **Simulated Configuration**

B. Comparison Points

We compare NVOverlay to five other mechanisms: (1) Software Undo Logging ("SW Logging"), (2) Software Shadow Paging ("SW Shadow"), (3) Hardware Shadow Paging ("HW Shadow"), (4) PiCL [59], and (5) PiCL running at L2 level (PiCL-L2). We briefly describe these mechanisms below.

Software Logging: Software generates and flushes an undo log entry before the first write. We assume that the software library tracks the write set, and flushes them at the end of an epoch. All NVM writes use barriers.

Software Shadow: Software tracks the write set and flushes dirty lines back at the end of each epoch. Software also maintains a persistent mapping table, which is updated at the end of an epoch. All NVM writes use barriers.

Hardware Shadow: We model hardware shadow paging using a three-version, cache line granularity shadow scheme similar to ThyNVM [65]. Hardware can overlap the persistence of the previous epoch with the execution of the current epoch. However, the centralized mapping table is updated synchronously.

PiCL: Uses hardware undo logging. Log entries are generated as in software logging. Hardware tracks dirty lines with a version-tagged, inclusive LLC. A tag walker periodically evicts dirty lines from previous epochs. We ignore the overhead of global epoch synchronization, and only focus on the data path.

PiCL-L2: A hypothetical design that functions the same as PiCL, except that tag walks are at the L2 level. We use this to estimate the performance of PiCL-style undo logging on large multicores without a monolithic and inclusive LLC.

For fairness of comparison, we assume all simulated designs, including the baselines, are equipped with a write-back DRAM buffer whose size can accommodate the entire working set. For our main experiments, the epoch size is set to 1M store uops [67]. Both PiCL and NVOverlay initiate tag walk (*ACS* in PiCL's terminology) after an epoch completes.

C. Benchmarks

We use benchmarks from the STAMP [51] suite and a set of data structure benchmarks for our evaluation. STAMP consists of memory-intensive applications that stress the data path and its transactional multicore synchronization model stresses the modified coherence protocol.

The data structure benchmarks consist of BTreeOLC [43], ARTOLC [42], red-black tree (`std::map`) and hash table (`std::unordered_map`). They represent workloads with large working sets. We run an insert-only workload with random keys to mimic bulk insertion into a database index.

All benchmarks spawn 16 worker threads, and are compiled with locks except BTreeOLC and ARTOLC. Threads execute until either the end of the program, or 100M instructions per-thread are reached (i.e. 1.6B total instructions).

VII. Experimental Results

A. Multicore Performance

Fig. 11 shows the wall-clock cycles for each workload using 16 worker threads. The results are normalized to an ideal NVM system with no snapshotting. For 9 out of 12 workloads, both NVOverlay and PiCL can fully overlap execution with snapshotting, incurring no cycle overhead. These results are also consistent with the original PiCL paper [59], in which PiCL was shown to perfectly overlap execution with persistence.

PiCL-L2, on the other hand, suffers from slightly slower execution, due to the smaller on-chip working set, which results in excessive evictions and log writes from the hierarchy. For `ssca2` and `kmeans`, PiCL-L2 runs 10% and 40% slower compared with NVOverlay and PiCL.

SW Logging and SW Shadow are both considerably slower than NVOverlay. The slowness is a natural consequence of writing both data and log entries synchronously.

Note that although NVOverlay can fully overlap the cost of persistence with execution in most cases, this does not contradict the observation that software schemes can be more than 2× worse. This is because NVOverlay is better at handling and distributing bursts of writes, which is the case for some workloads (e.g. shifting existing elements after locating a B+Tree leaf node). For example, our evaluation shows that in `B+Tree` workload, out of 11,778,311 total NVM data write requests, 11,503,974 (97.7%) of them are generated by the coherence protocol. NVOverlay evenly distributes these writes across the execution.

HW Shadow is moderately slower than NVOverlay. In all but `B+Tree` and `ART`, HW Shadow is at most 3× slower than NVOverlay. We attribute this to its ability to overlap data persistence with execution. It has to, however, synchronously update the mapping table at the end of an epoch to avoid corrupting the table in the next epoch.

B. Write Amplification

Fig. 12 shows the results of write amplification in terms of bytes written to the NVM device. We measure both data and metadata writes. For PiCL and PiCL-L2, we assume each log entry takes 72 bytes (64B data + 8B address tag). For HW

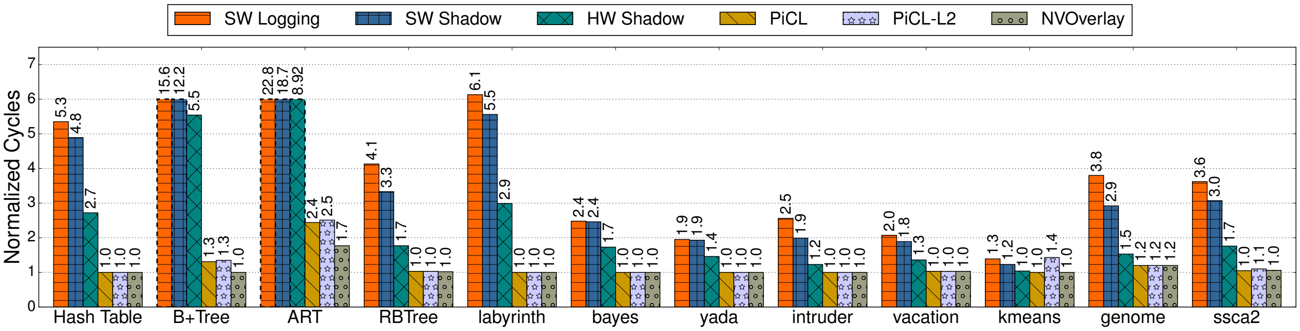


Fig. 11: **Normalized Cycles** – 16 worker threads. All numbers are normalized to baseline execution without snapshotting.

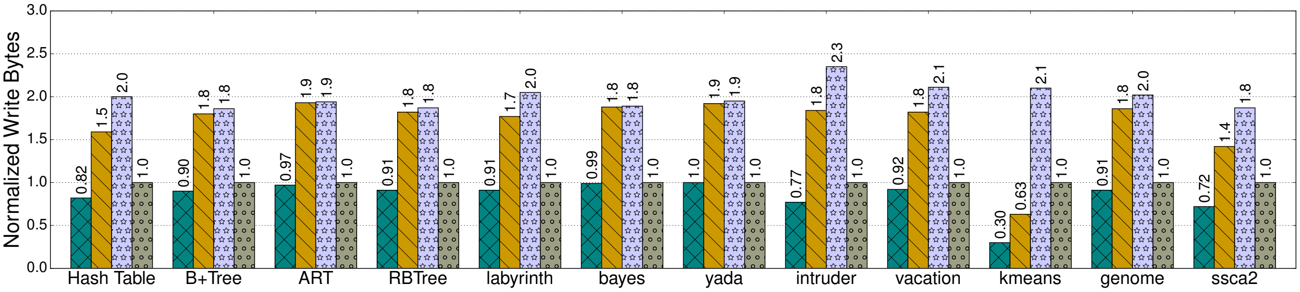


Fig. 12: **Write Amplification (Bytes of Data)** – 16 worker threads. All numbers are normalized to NVOverlay.

Shadow and NVOverlay, we track the number of eight byte writes performed on the radix tree mapping table. All numbers are normalized to NVOverlay.

As stated in Section II, both HW Shadow and NVOverlay demonstrate lower write amplification than logging. This is an expected result, since PiCL and PiCL-L2 need to write two full cache lines for each cache eviction instead of one. Overall, PiCL writes 1.4×–1.9× more data than NVOverlay. For PiCL-L2, write amplification is higher, ranging between 1.8×–2.3×. This is caused by the smaller on-chip working set, as we discussed in the previous section.

In 5 out of 12 workloads, NVOverlay incurs more than 10% writes than Shadow Paging. This is because NVOverlay's coherence protocol issues a write to the NVM when the ownership of a cache line is transferred from the upper level of the hierarchy to the LLC. The negative effect of such write backs can become significant, as we see in `kmeans`, where 70% fewer writes are issued to the NVM for HW Shadow, and 37% fewer for PiCL. PiCL-L2, on the other hand, issues 2× more writes to the NVM compared with NVOverlay, due to extra log writes.

Further studies of `kmeans` reveal that only 896,837 writes are issued from the LLC when simulating HW Shadow, while 3,087,987 writes (3.4× more) are issued from the L2 when simulating NVOverlay. Among these 3 million writes, 2,413,754 are caused by L2 capacity miss evictions, 668,951 by load-downgrade, 4,994 by store-eviction, and 288 by other events. From these numbers, we conclude that `kmeans` suffers from L2 thrashing by writing a large portion of data it fetched into L2, and later forced to evict them on capacity misses. This explains why `kmeans` favors LLC-based hardware schemes.

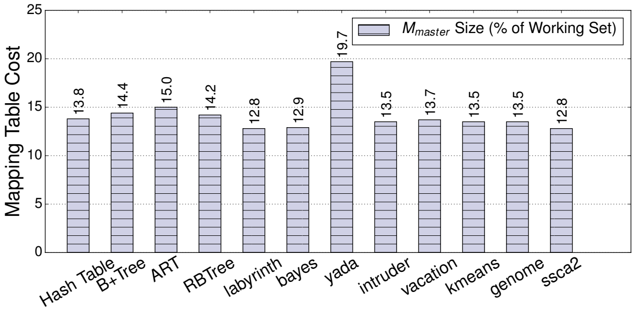


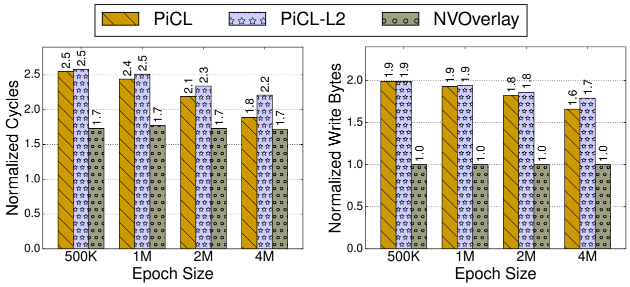
Fig. 13: **Persistent Mapping Metadata Cost** – All numbers are percentage of working set size.

Fortunately, the larger write amplification does not translate to higher execution time for NVOverlay.

C. Persistent Metadata Overhead

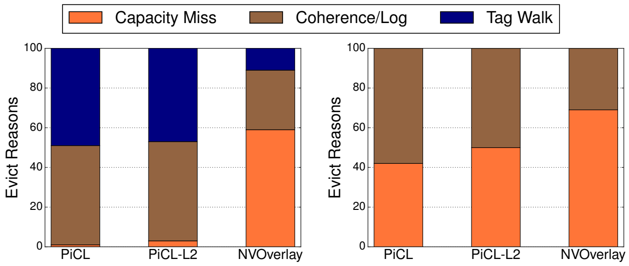
Fig. 13 presents the size comparison between the Master Mapping Table (M_{master}, see Section V-C) and the write working set. We define the write working set as the total amount of data mapped by M_{master}.

The ratio between working set size and M_{master} size is rather stable across different runs. In all workloads except `yada`, the metadata cost is between 12.8%–15.1% of the working set size. This result is consistent with the property of radix trees: In a perfectly populated tree, each 8 byte pointer in a leaf node can map a 64 byte cache line, achieving a theoretical lower bound of 12.5%. Our results show that in most cases, NVOverlay can achieve an almost optimal use of mapping table storage, thanks to the locality of computation.



(a) Normalized Cycles (b) Normalized Writes

Fig. 14: **Sensitivity to epoch size (ART benchmark)** – Cycles are normalized to baseline; Writes are normalized to NVOverlay.



(a) With Tag Walker (1M Epoch) (b) Without Tag Walker

Fig. 15: **Evict Reason Decomposition** – Workload is ART.

As for yada, further investigation shows that each inner page in the table only maps 18.14 pages (3.54% of total slots) on average, implying low occupancy of inner pages. In contrast, 93.66% of leaf page slots map a cache line, suggesting that locality in small address ranges are still maintained.

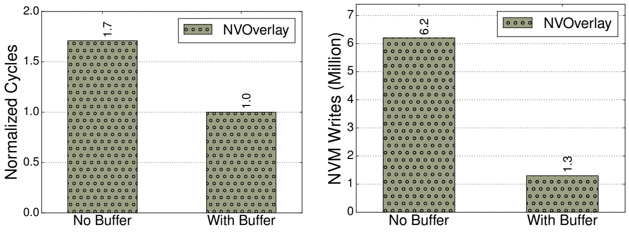
D. Sensitivity Study

1) Epoch Size: We study the effect of varying epoch sizes (and hence tag walk frequency) by simulating PiCL, PiCL-L2 and NVOverlay on ART, with epoch sizes ranging from 500K to 4M. Results are shown in Fig. 14a (cycles) and Fig. 14b (write amplification).

Cycles: Both NVOverlay and PiCL-L2 are insensitive to epoch size change. This could be explained by the fact that most evictions are caused by coherence load-downgrade (2,105,356, 29.7%) and L2 capacity miss eviction (4,194,012, 59.1%), while tag walks (792,255, 11.2%) only marginally contribute to total write bandwidth. PiCL, on the other hand, performs better under long epochs, since around half of the evictions are generated by tag walk evictions (6,086,088, 50%), which are necessary to commit a previous epoch.

Write Amplification: As epoch size increases, write amplification of both PiCL and PiCL-L2 steadily drops, as a result of reduced frequency of tag walks. Besides, since dirty cache lines can survive longer in the cache without being forced out, less log entries are generated. As epoch size increases from 500K to 5M, write amplification drops 11.0% and 15.9% for PiCL and PiCL-L2, respectively.

2) Tag Walker: To evaluate the effect of tag walker on performance, we simulate PiCL, PiCL-L2 and NVOverlay on ART, with and without the tag walker. Results are presented in Fig. 15 as a decomposition of evict reasons.



(a) Normalized Cycles (b) NVM Writes

Fig. 16: **Reducing Writes with OMC Buffer** – Workload is ART.

Both PiCL and PiCL-L2 are heavily dependent on the tag walker for making progress. As in Fig. 15a, more than 47% of total write requests are generated by tag walks, indicating that the tag walker might become a performance bottleneck on larger caches for PiCL and PiCL-L2. NVOverlay, by contrast, writes back dirty lines mainly by cache coherence and capacity miss eviction, which is distributed evenly during the execution. The tag walker only contributes to around 11% of total evictions. The efficiency of the tag walker, therefore, has limited effect on NVOverlay, as shown in Fig. 15b.

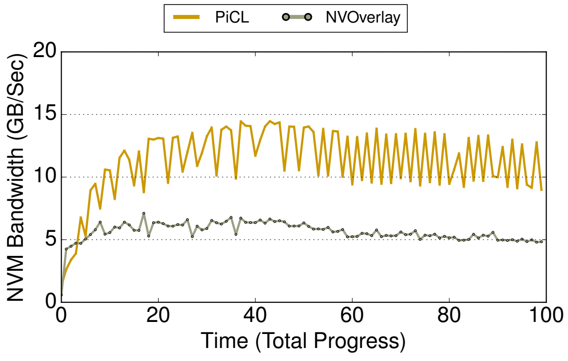
3) OMC Buffer: We evaluate the persistent OMC buffer proposed in Section IV-E by simulating NVOverlay on ART, with and without the buffer. The evaluation has only one epoch throughout the execution to stress-test the buffer's ability to absorb redundant write backs (i.e., those generated on the same address and in the same epoch) from the hierarchy. We use a buffer that has the same configuration as the simulated LLC, with the expectation that it would further reduce NVM write traffic as if NVOverlay were built on the LLC. Results are in Fig. 16.

As shown in Fig. 16a, the OMC buffer improves performance by 41%. Fig. 16b further reveals that the performance improvement is a result of reduced writes, proving its effectiveness. Out of total 7,136,893 write requests, 5,336,687 hit the buffer, achieving a hit rate of 74.8%.

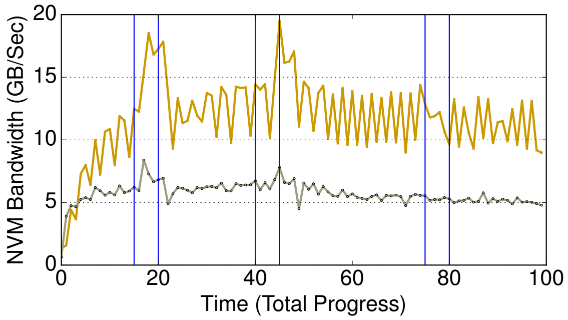
E. Bandwidth

To evaluate bandwidth benefits, we simulate NVOverlay and PiCL on BTree, and measure NVM write bandwidth. Results are presented in Fig. 17. Fig. 17a shows bandwidth over time during the entire simulation using the default epoch size. NVOverlay demonstrates two clear advantages: (1) Average bandwidth consumption is significantly lower than PiCL; (2) Peak bandwidth and overall fluctuation is also lower, indicating better scalability since more components can be supported on a fixed bandwidth budget. We attribute this to the fact that NVOverlay's version coherence "amortizes" version write back bandwidth over regular execution, while PiCL must evict dirty lines with tag walk, creating bandwidth surges at epoch boundaries.

Fig. 17b shows bandwidth over time when epochs occur in short but localized bursts. We use this to mimic time-travel



(a) BTree, 1M Default Epoch



(b) BTree, Bursty Epoch

Fig. 17: **NVM Write Bandwidth Time Series**

debugging, where programmers may manually start new epochs around suspicious code segments. Three "bursty" intervals are present, marked by blue vertical lines. From left to right, the size of epochs in these bursty intervals are 1K, 10K, and 100K, respectively. The figure demonstrates that when epoch size is moderately small (100K), both schemes work as usual. With extremely small epochs (1K, 10K), however, NVOverlay still sustains relatively lower bandwidth, while PiCL observes 50% more traffic due to frequent log generation.

VIII. Additional Related Work

We now discuss some additional related work beyond what we already covered in Section II. Journaling has long been implemented on conventional file systems for crash recovery [64]. Before an operation commits, the changes to its metadata[4] are first flushed to the journal area of the disk, which can be replayed after a crash for the sake of recovery. Journaling is related to a log-structured file system (LFS) [66], which treats the entire file system as a large journal. Under LFS, there is no fixed "home location" for data and metadata: a mapping table is used to locate an item given its logical address, and this mapping table is also journaled to the log. NVOverlay's backend (MNM) works similarly to LFS, with a key difference being that there are *multiple* journal objects (rather than one), each representing an epoch's working data.

Regarding using NVM to improve file systems, one of the earliest NVM file system designs—BPFS [9])—treated the entire file system as a forest of B+Trees. With BPFS, each update operation creates a new snapshot of the file system via Copy-on-Write (CoW) and atomic pointer swings. Similarly, Jacob [25] describes a flash-based design that tracks snapshots by "chaining" mapping entries of the same address together through a combination of a customized FTL that performs CoW, along with firmware-maintained mapping table. These snapshots can be retrieved later for crash recovery.

In contrast with these previous designs that target file system crash recovery, our goal in this paper of supporting high-frequency snapshotting of the full address space in DRAM-based main memory involves several additional challenges. First, the snapshotting frequency necessary to simply support crash recovery is orders of magnitude slower than what we are targeting with NVOverlay, thereby making it much easier to avoid performance bottlenecks. Second, file systems (and flash storage) already contain explicit metadata to track the location of data, which is not the case for DRAM main memory. Enabling this ability to distinguish the data from separate multiversioned snapshots in DRAM and caches while still running at full speed is a major aspect of our NVOverlay design. Third, file systems involve an explicit software interface for write operations, unlike CPU writes to main memory (which are performed by regular write instructions, at very high frequencies). As described earlier, NVOverlay uses Coherent Snapshot Tracking (CST) and Multi-snapshot NVM Mapping (MNM) to overcome these challenges.

IX. Conclusions

In this paper, we have presented and evaluated a novel hardware technique (NVOverlay) for snapshotting to NVM the full address space of an unmodifed parallel application running on a large-scale multiprocessor. By leveraging page overlays to support lazily persisting a number of cache-resident checkpoints, NVOverlay achieves significantly better performance than hardware shadow paging on a number of benchmarks. Compared with hardware logging, NVOverlay significantly reduces the amount of write amplification (by roughly a factor of two) since it avoids writing both a log and dirty data to the NVM. When NVM bandwidth becomes precious (e.g., in the ART benchmark), this bandwidth savings can have a signficant performance impact. By supporting snapshotting in an efficient and scalable fashion, NVOverlay makes both crash recovery and the frequent checkpointing that may occur in checkpoint-based debugging practical for large-scale parallel applications.

References

[1] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," *ISCA*, 2004.

[2] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: A high-performance latch-free range index for non-volatile memory," *VLDB*, January 2018.

[3] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind logging," *VLDB*, 2016.

[4] J.-L. Baer and W.-H. Wang, "On the inclusion properties for multi-level cache hierarchies," *Computer Architecture News*, 1988.

[5] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," *OOPSLA*, 2016.

[4] In some versions of journaling, the changes to file data are also logged.

[6] M. N. Bojnordi and E. Ipek, "Pardis: A programmable memory controller for the ddrx interfacing standards," *ISCA*, 2012.

[7] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *OOPSLA*, 2014.

[8] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," *ASPLOS*, 2019.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," *SOSP*, 2009.

[10] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," *SPAA*, 2018.

[11] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," *ASPLOS*, 2015.

[12] A. M. Devices, "Amd64 architecture programmer's manual volume 2: System programming," 2006.

[13] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine," *SIGMOD*, 2013.

[14] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," *HPCA*, 2016.

[15] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *STOC*, 1986.

[16] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, 2002.

[17] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, 2008.

[18] J. Gaur, M. Chaudhuri, and S. Subramoney, "Bypass and insertion algorithms for exclusive last-level caches," *ISCA*, 2011.

[19] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, "Pisces: A scalable and efficient persistent transactional memory," *ATC*, 2019.

[20] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," *MICRO*, 2019.

[21] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," *EuroSys*, 2017.

[22] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda, "Log-structured non-volatile main memory," *ATC*, 2017.

[23] D. Hwang, W. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," *FAST*, 2018.

[24] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ASPLOS*, 2016.

[25] B. Jacob, "The 2 petaflop, 3 petabyte, 9 tb/s, 90 kw cabinet: a system architecture for exascale and big data," *IEEE Computer Architecture Letters*, 2015.

[26] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer, "Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies," *MICRO*, 2010.

[27] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," *MICRO*, 2018.

[28] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," *ISCA*, 2018.

[29] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," *HPCA*, 2017.

[30] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," *MICRO*, 2015.

[31] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: Reducing software overhead in file systems for persistent memory," *SOSP*, 2019.

[32] R. Kateja, N. Beckmann, and G. Ganger, "Tvarak: software-managed hardware offload for redundancy in direct-access nvm storage," *ISCA*, 2020.

[33] R. Kateja, A. Pavlo, and G. Ganger, "Vilamb: Low overhead asynchronous redundancy for direct access nvm," *Carnegie Mellon University Parallel Data Lab*, 2019, technical report CMU-PDL-20-101.

[34] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," *ASPLOS*, 2002.

[35] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: Exploiting nvram in write-ahead logging," *ASPLOS*, 2016.

[36] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," *MICRO*, 2016.

[37] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," *ASPLOS*, 2016.

[38] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, 1978.

[39] P.-r. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-performance concurrency control mechanisms for main-memory databases," *VLDB*, 2011.

[40] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: write optimal radix tree for persistent memory storage systems," *FAST*, 2017.

[41] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent dram indexes to persistent-memory indexes," *SOSP*, 2019.

[42] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," *ICDE*, 2013.

[43] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The art of practical synchronization," *DaMoN*, 2016.

[44] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," *ASPLOS*, 2017.

[45] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "ido: Compiler-directed failure atomicity for nonvolatile memory," *MICRO*, 2018.

[46] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," *HPCA*, 2018.

[47] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "Pmtest: A fast and flexible testing framework for persistent memory programs," *ASPLOS*, 2019.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *PLDI*, 2005.

[49] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum, "Towards practical default-on multi-core record/replay," *ASPLOS*, 2017.

[50] H. McIntyre, S. Arekapudi, E. Busta, T. Fischer, M. Golden, A. Horiuchi, T. Meneghini, S. Naffziger, and J. Vinh, "Design of the two-core x86-64 amd bulldozer module in 32 nm soi cmos," *IEEE journal of solid-state circuits*, 2011.

[51] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," *2008 IEEE International Symposium on Workload Characterization*, 2008.

[52] A. Miraglia, D. Vogt, H. Bos, A. Tanenbaum, and C. Giuffrida, "Peeking into the past: Efficient checkpoint-assisted time-traveling debugging," *ISSRE*, 2016.

[53] A. Mirhosseini, A. Agrawal, and J. Torrellas, "Survive: Pointer-based in-dram incremental checkpointing for low-cost data persistence and rollback-recovery," *IEEE Computer Architecture Letters*, 2017.

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

[55] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," *TRIOS*, 2013.

[56] D. Mulnix, "Intel xeon processor scalable family technical overview," 2017.

[57] M. Nam, H. Cha, Y.-R. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," *FAST*, 2019.

[58] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," *SIGMOD*, 2015.

[59] T. M. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," *MICRO*, 2018.

[60] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging," *MICRO*, 2019.

[61] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+redo logging for persistent memory systems," *HPCA*, 2018.

[62] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory," *SIGMOD*, 2016.

[63] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *ISCA*, 1984.

[64] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems." *ATC*, 2005.

[65] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory

systems," *MICRO*, 2015.

[66] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM TOCS*, 1992.

[67] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ISCA*, 2013.

[68] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for NVRAM," *ADMS*, 2015.

[69] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-atomic slotted paging for persistent memory," *ASPLOS*, 2017.

[70] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable ssd," *OSDI*, 2014.

[71] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, and T. Chilimbi, "Page overlays: An enhanced virtual memory framework to enable fine-grained memory management," *ISCA*, 2015.

[72] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "Memzip: Exploring unconventional benefits from memory compression," *HPCA*, 2014.

[73] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," *MICRO*, 2017.

[74] S. M. Srinivasan, S. Kandula, C. R. Andrews, Y. Zhou *et al.*, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," *ATC*, 2004.

[75] M. E. Thomadakis, "The architecture of the nehalem processor and nehalem-ep smp platforms," *Texas A&M Tech Report*, 2011.

[76] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," *FAST*, 2011.

[77] N. Viennot, S. Nair, and J. Nieh, "Transparent mutable replay for multicore debugging and patch validation," *ASPLOS*, 2013.

[78] T. Wang, J. J. Levandoski, and P. Larson, "Easy lock-free indexing in non-volatile memory," *ICDE*, 2018.

[79] Z. Wang, M. A. Kozuch, T. C. Mowry, and V. Seshadri, "Multiversioned page overlays: Enabling faster serializable hardware transactional memory," *PACT*, 2019.

[80] C. Warner, D. Robinson, J. Wastlick, M. Schroeder, and J. Moy, "Noninclusive cache systems and methods," 2014, uS Patent 8,661,208.

[81] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye, "Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory," *ISCA*, 2020.

[82] S. Wu, F. Zhou, X. Gao, H. Jin, and J. Ren, "Dual-page checkpointing: An architectural approach to efficient data persistence for in-memory applications," *ACM Trans. Archit. Code Optim.*, 2019.

[83] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, "An empirical evaluation of in-memory multi-version concurrency control," *VLDB*, 2017.

[84] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," *ASPLOS*, 2019.

[85] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," *FAST*, 2016.

[86] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," *FAST*, 2015.

[87] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," *ASPLOS*, 2015.

[88] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," *ASPLOS*, 2015.

[89] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," *MICRO*, 2013.

[90] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes," *MICRO*, 2018.

[91] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," *OSDI*, 2018.

[92] P. Zuo, Y. Hua, and Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," *MICRO*, 2019.