Distributed Logless Atomic Durability with Persistent Memory

Siddharth Gupta
EcoCloud, EPFL
siddharth.gupta@epfl.ch

Alexandros Daglis
Georgia Institute of Technology
alexandros.daglis@cc.gatech.edu

Babak Falsafi
EcoCloud, EPFL
babak.falsafi@epfl.ch

ABSTRACT

Datacenter operators have started deploying Persistent Memory (PM), leveraging its combination of fast access and persistence for significant performance gains. A key challenge for PM-aware software is to maintain high performance while achieving atomic durability. The latter typically requires the use of logging, which introduces considerable overhead with additional CPU cycles, write traffic, and ordering requirements. In this paper, we exploit the data multiversioning inherent in the memory hierarchy to achieve atomic durability without logging. Our design, LAD, relies on persistent buffering space at the memory controllers (MCs)—already present in modern CPUs—to speculatively accumulate all of a transaction's updates before they are all atomically committed to PM. LAD employs an on-chip distributed commit protocol in hardware to manage the distributed speculative state each transaction accumulates across multiple MCs. We demonstrate that LAD is a practical design relying on modest hardware modifications to provide atomically durable transactions, while delivering up to 80% of ideal—i.e., PM-oblivious software's—performance.

CCS CONCEPTS

• **Computer systems organization → Processors and memory architectures**; • **Information systems →** *Storage class memory*; *Phase change memory*.

KEYWORDS

Persistent Memory, Atomic Durability, Logging, Atomicity

ACM Reference Format:
Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. 2019. Distributed Logless Atomic Durability with Persistent Memory. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358321

1 INTRODUCTION

Persistent memory (PM) is making inroads into datacenters: battery-backed DRAM is already used in production [21] and emerging products based on non-volatile memory technologies, such as Intel's 3D XPoint [12], are further popularizing PM's deployment. PM promises performance gains for data-intensive applications by replacing slow disk-based I/O with fast memory accesses to byte-addressable memory.

Making effective use of PM requires software to be specially designed for crash consistency. For the popular programming abstraction of transactions, crash consistency requires a transaction to be atomically durable. Atomic durability guarantees that upon a power failure, either all or none of a transaction's updates are made durable. A key challenge for software is achieving atomic durability without significantly hurting the performance boost gained by replacing conventional storage with PM. The mainstream approach for atomic durability is the use of logs, with most instances relying on software-based write-ahead logging [15, 28, 42] and a number of research proposals accelerating logging in hardware [20, 31, 37]. While its strengths and weaknesses vary with the specific implementation, logging in general introduces programmability hurdles and notable performance overheads—excess CPU cycles, PM writes, and ordering constraints—especially for short transactions.

In this paper, we leverage the multiple versions of data that are inherently present in the memory hierarchy to eschew logging. PM always holds the data values valid before a transaction's start. During the transaction's runtime, all of its updates can be collected in a speculative staging area higher up in the memory hierarchy and atomically committed to PM as soon as the transaction commits. A critical requirement for this high-level idea to work is that the staging area should also be persistent, to guarantee atomic propagation of the transaction's updates to PM in case of power failure. We argue that the request queues of the memory controllers (MCs) are excellent candidates to implement such a staging area. Not only are the battery-backing requirements modest because of the small capacity of these queues, but battery-backed MCs are already available on the latest server CPUs [36, 39].

We introduce LAD (Logless Atomic Durability), a hardware mechanism that exposes the familiar interface of a transaction to software and guarantees that all of the transaction's updates propagate atomically to PM without the use of logging. LAD buffers all updates in the persistent MC queues while a transaction is running and atomically commits them to PM when the transaction ends. The general concept LAD relies on to eschew logging, namely the use of a persistent staging area within the memory hierarchy to collect data before atomically committing to PM, has been introduced before by Kiln [46]. Unlike Kiln, LAD (i) limits persistence requirements to the MCs without any LLC modifications and (ii) supports scalable memory hierarchies that don't feature a point of centralization, like a unified LLC. As the memory hierarchy of modern server CPUs is distributed (e.g., NUCA LLC and multiple MCs), we address the critical challenge of managing speculative state that is also distributed, because a single transaction's updates can touch data residing in different memory locations. LAD employs a variant of the two-phase commit protocol implemented in

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358321

hardware to handle distributed speculative state and make atomic commit decisions.

Prior work on atomic isolation involves management of similar distributed speculative state. However, the proposed techniques are not directly applicable in the case of atomic durability, as the concepts of failure and recovery qualitatively differ in the two contexts. At a high level, failure in the context of atomic durability is abrupt as a result of a power loss, so at any given time instance, all state necessary for recovery needs to be persistent and consistent.

Our main contribution is the design of a hardware mechanism for atomic durability that contains all hardware extensions to the L1D caches and MCs, without requiring any on-chip caches to be persistent. LAD obviates the software overhead of log creation and—in the common case—avoids any form of logging altogether. Consistent with technological trends, the only non-volatile on-chip components LAD relies on are MCs, which are already battery-backed in recent server CPUs. Furthermore, we detail an implementation of a distributed hardware-managed protocol for atomic decisions that is robust in face of system crashes and power failures.

The rest of the paper is organized as follows. In §2, we discuss prior work and argue that the recent technology trend of battery-backed MCs introduces new design opportunities for high-performance atomic durability. We describe LAD's design and implementation in §3 and §4, respectively. We continue with our methodology (§5) and evaluation (§6), and conclude in §7.

2 BACKGROUND AND RELATED WORK

PM is a broad term describing byte-addressable persistent memory, including both non-volatile memory technologies (e.g., Intel's 3D XPoint [12]) and battery-backed DRAM [21]. PM is gaining momentum as it promises significant performance gains for data-intensive applications, offering the persistence property—traditionally only attainable with I/O operations to storage devices—at memory latency [21, 24].

To benefit from PM's persistence property, software has to follow certain PM-specific contracts to be crash-consistent, namely guarantee that PM's contents allow the application to recover to a valid operational state after a system crash. A key source of complexity in designing crash-consistent software arises from the fact that the order in which data updates reach PM is different from the memory order observed by the CPU (i.e., the CPU's memory model) and completely transparent to the software. For performance reasons, CPUs feature a deep cache hierarchy above the memory, which is typically volatile even in systems that deploy PM. Unlike data residing in PM, all contents of the volatile cache hierarchy are lost upon a system crash. With write-back caches, data is only written to PM upon eviction; therefore, the order in which updates reach PM can drastically differ from program order. As a result, unless special care is taken in software to deal with this issue, unordered data updates in PM lead to corrupted application state when the system reboots after a crash.

To give software control over the order of update propagation to PM, modern server CPUs extend their ISA with special *flush* instructions that explicitly write a target cache block back to PM. Once the cache block is made persistent, the memory controller (MC) acknowledges the flush's completion. For example, Intel's processors feature the `clwb` instruction [14] for that purpose. To remove the high latency of writing through to PM from the critical path, Intel recently made the MCs supporting PM persistent [36, 39]. This enhancement allows flushed cache blocks to attain durability status as soon as they reach an on-chip MC and the `clwb` message to be immediately acknowledged by the MC.[1]

2.1 Atomic Durability

PM-aware software uses ISA extensions, such as `clwb`, to force selective propagation of data updates to PM and thus control the order of persistent updates. However, control on ordering alone is not a flexible enough tool for software, which requires higher-level programming abstractions, such as that of a transaction: a block of instructions with effects that either occur atomically, or not at all. Among a range of existing transaction semantics, ACID [6, 11] is a popular instance providing multiple desirable properties together.

In this paper, we focus on a subset of the ACID properties, atomic durability, which is the guarantee that upon a system crash, either all of a transaction's updates are made persistent, or all collectively discarded. As existing mechanisms that offer atomic durability either significantly hamper performance or require extensive hardware modifications, we introduce a practical alternative that preserves PM's high performance. Our proposed solution only contributes toward improved atomic durability performance. LAD can be combined with a concurrency control mechanism to attain full ACID transactions; in fact, the implementation we detail in this paper is readily combinable with conventional locking mechanisms.

Prior work pursuing the same goal of high-performance atomic durability can be broadly classified into two categories: logging and hardware versioning. We briefly cover proposals from both categories, highlighting the inherent overheads in all forms of logging, which hardware versioning alleviates. Then, in §2.2, we underline the salient differences of our proposed design for logless atomic durability from Kiln [46], the most relevant prior proposal leveraging hardware versioning.

2.1.1 Logging. Write-ahead logging [28] creates explicit duplicate versions of data in the form of logs that are made persistent before any in-place updates. In case of a crash, these logs are used to restore data to a consistent state. Common logging mechanisms include undo and redo logging, which log original data and updates respectively. Logging is by far the most popular mechanism for atomic durability, and is implemented in either software or hardware.

Software logging uses flush instructions such as `clwb` to write the logs to PM before any in-place data updates become persistent [5, 6, 13, 15, 17, 22, 25, 30, 42, 43]. Kamino-TX [27] instead maintains a replica of the dataset to serve as an undo log, thus removing logging from the critical path. Transactions directly apply in-place updates and the replica is asynchronously updated in the background.

Hardware logging techniques introduce hardware support to improve the performance of logging. In some instances, logs are generated by the CPU, and special hardware support accelerates log management after their creation (i.e., writing the logs back to PM and preserving correct ordering between logs and in-place

[1] Existing MCs are not truly persistent, but come with enough battery backup to flush their queues' contents back to memory upon power failure; for our purposes, the effect is the same. We therefore refer to MCs with that feature as *persistent*.

updates) [6, 8, 18, 23, 26, 38]. In other proposals, logging is offloaded to dedicated hardware altogether [7, 19, 20, 31, 37].

All forms of logging incur overhead related to log creation and management. For software logging, the CPU executes a potentially significant number of additional instructions per transaction. The log also needs to be serialized and made persistent before any in-place updates. In the case of undo logging, dynamic transactions for which addresses of updates are not known in advance incur multiple explicitly ordered logging actions, hurting the CPU's throughput [25, 43]. In redo logging, in-place updates are delayed with respect to the log's creation, thus all reads need to check the log first to guarantee that the latest data values are always accessed [25, 34, 43].

In every case, writing logs to PM increases bandwidth demands. The cost of logging is disproportional to its utility. As crashes are rare, logs are rarely used for recovery; in the common case, they are simply erased shortly after their creation. Last but not least, logging may also require non-trivial changes in application code, introducing a programmability burden, especially when implemented without the use of high-level library primitives [15, 42].

2.1.2 Hardware Versioning. Logging explicitly creates a second version of a transaction's updated data, to enable rollback in case of a crash. Kiln [46] leverages the data multiversioning inherent in multi-level memory hierarchies to obviate logging and its associated overheads. Fig. 1a demonstrates a conventional system with PM, assuming no battery-backed MCs, where all on-chip components are volatile. In such a configuration, achieving atomic durability through logging requires serialized log writes to off-chip memory, incurring a high-latency operation. To tackle this challenge, Kiln brings the persistent domain closer to the CPU, by replacing the default SRAM-based LLC with a persistent memory technology (STT-RAM) and leverages this quickly accessible persistent LLC to remove logging altogether. While a transaction runs, its updates are accumulated in the persistent LLC, marked as speculative (Fig. 1b). As soon as the transaction ends, its updates are instantaneously committed to PM by simply clearing their speculative markers. Upon a post-crash reboot, the persistent LLC discards cache blocks found in the speculative state.

Overall, Kiln addresses the shortcomings of logging, but makes two assumptions that generally don't hold in modern server CPUs: that the LLC is persistent and centralized. We next elaborate on these limiting assumptions and how LAD addresses them.

2.2 Distributed Persistent MCs

Speculatively buffering updates within a quickly accessible persistent domain is an effective approach to achieve atomic durability that tackles the performance and programmability overheads of logging. Inspired by the persistent nature of MCs in modern server CPUs, we investigate whether the persistent MC queues can be similarly used as a staging area for speculative updates of atomically durable transactions, as demonstrated in Fig. 1c.

Our proposed design, LAD, addresses the challenge of managing distributed speculative state to drive decisions for atomically committing a transaction to PM, a need that fundamentally arises from the presence of several MCs in servers. MCs can be distributed even in single-socket systems. For example, Intel's mesh-based CPUs place MCs in two physically disjoint on-chip locations (IMC tiles)

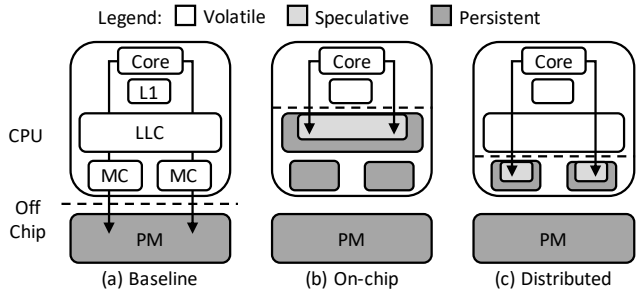


Figure 1: Persistent domain and staging area of speculative buffering for atomic durability. The dashed line marks the persistence boundary.

[1], while each AMD EPYC socket is a multi-chip package comprising four chips, each with its own northbridge/MC [2]. Such MC distribution significantly affects the design of an atomic commit protocol. Even though messages from a CPU core may reach the MCs at different times, all MCs must make decisions in unison to maintain atomicity. A crash may result in partial delivery of these messages. Thus, handling atomic durability fundamentally requires treating the involved on-chip resources (i.e., the CPU cores and all MCs) as a distributed system.

The complication of managing distributed state is not an artifact of using MCs as a staging area. Given the ubiquitous presence of distributed LLCs (NUCA) in server CPUs [3], placing the staging area in the LLC, as per Kiln's proposal, results in distributed speculative state too. Even if future servers adopt persistent NUCA LLCs, leveraging them for atomic durability will still require a distributed commit protocol.

Finally, LAD shows that a persistent LLC is not necessary for high-performance atomic durability. Limiting the persistent domain to the MCs, a feature readily available in modern server CPUs, is sufficient. In other words, extending the persistent domain beyond the MCs (e.g., to the LLC) is a largely unnecessary modification with diminishing returns.

2.3 Distributed State in Other Contexts

Prior work on persist barriers, ordering update flushes from the LLC to PM, identified the need for consensus when the LLC is distributed [18]. The authors developed a distributed consensus protocol to synchronize flush ordering across LLC slices. In contrast, LAD's protocol is developed to achieve atomicity, a stronger guarantee than ordering.

The concepts underlying LAD's proposed mechanism for atomic persistence also bear high-level similarities with the broad range of work on hardware transactional memory. While transactional memory implementations vary greatly, they are all concerned with management of speculative state to achieve atomicity in the context of isolation. We find more relevance between LAD and mechanisms that manage distributed speculative state to make atomic decisions in hardware, which are not necessarily framed in the context of transactional memory. For instance, on-chip hardware implementations of distributed protocols resembling the well-known two-phase

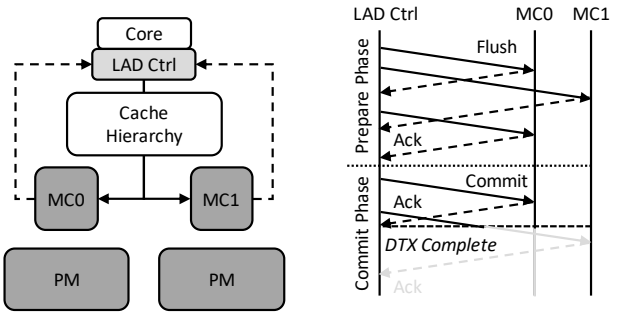
commit protocol [41] have been deployed to achieve sequential consistency using update-based coherence [45] or to atomically commit blocks of memory operations to improve the performance of shared-memory multiprocessing [4, 32, 33]. Although implementation aspects differ between these proposals and LAD, they all share the high-level goal of bringing consensus among multiple agents to make atomic decisions, and protocol operation at a steady state follows similar stages, as we will describe in §3.2. The key difference of prior proposals and LAD is that such mechanism is employed to achieve atomic durability rather than isolation. This qualitative difference introduces new design limitations and challenges, such as identifying the most appropriate on-chip component to manage speculative atomically durable state (i.e., the MCs—§2.2), preserving the atomic durability trait for arbitrarily long transactions causing speculative buffer overflows (§3.3), and preserving or recovering atomicity in face of system crashes and power failures (§3.4).

A particular aspect that differentiates LAD from other protocols for distributed state management to make atomic decisions is the failure model and system recovery after failure. In mechanisms designed for atomic isolation, such as [4, 32, 33], a transaction fails when a race with another transaction is detected and speculative changes have to be rolled back. Due to the target domain's nature, there is significant flexibility in how and where to maintain state, how failure is detected, which of the conflicting transactions to abort, whether to react eagerly or lazily, etc. In the context of atomic durability, failure is not associated with concurrency, but with abrupt system failure because of a system crash or a power loss, which introduces new types of protocol failures (e.g., power loss precludes some MCs from receiving a broadcast message). Hence, the mechanism has to maintain state that is resilient—persistent and consistent—to failure at all times and enables recovery to a fully consistent state when the system is brought back to normal operation at a later point in time. The diverging set of assumptions in these two different contexts leads to different design choices. For example, the frequency of failures in the context of atomic durability is extremely low compared to failures in concurrency control, clearly tipping the scale toward optimistic rather than pessimistic mechanisms. A clear demonstration of this guideline is our choice of undo logging as a fallback mechanism (§3.3).

3 LAD DESIGN

3.1 Overview

LAD builds on the existence of an on-chip persistent domain accessible by the CPU at low latency, the MCs, which can be used as a staging area where a transaction's updates are collected speculatively until they can be atomically committed to PM. Such mechanism replaces logging as the means to achieving atomic durability. LAD's high-level design includes CPU-side and MC-side controllers, which interact to deliver atomically durable transactions (DTX). In this section, we refer to the CPU-side controller as the *LAD controller*. The LAD controller manages the stages of each DTX as it executes on the CPU and interacts with the MC-side controllers. Because server-class CPUs feature several MCs, a single DTX's updates will be spread across multiple MCs. Therefore, achieving atomic durability requires the coordination of the LAD controller with all of the MCs, each of which holds a fraction of the DTX's



(a) System architecture. (b) Protocol timeline.

Figure 2: Distributed commit protocol.

speculative state. LAD employs a distributed protocol to handle that distributed speculative state and guarantee each DTX's atomic durability. This section details LAD's design.

3.2 Distributed Commit Protocol

LAD's distributed commit mechanism is inspired by the two-phase commit (2PC) protocol [41], commonly implemented in software for distributed systems. LAD's adaptation of 2PC is implemented in hardware, to handle the distributed speculative state each DTX accumulates at the MCs during its execution. The LAD controller is collocated per CPU core, controls the protocol's flow and directly interacts with the MCs, as shown in Fig. 2a. §4 describes a concrete LAD controller implementation.

Fig. 2b demonstrates the protocol's phases and exchanged messages. A DTX is divided into two phases: `Prepare` and `Commit`. In the `Prepare` phase, while a CPU executes a DTX, the LAD controller flushes all of the CPU's writes that reside in the cache hierarchy and match PM addresses to the MCs. The flushed updates carry a unique identifier, comprising a thread ID and a private-per-thread DTX ID, incremented with every new DTX. All of the DTX's updates contained in the MC queues are marked speculative and are not written back to PM. When the CPU reaches the DTX's end, the LAD controller waits for all outstanding flushes to be acknowledged by the MCs before proceeding to the `Commit` phase.

In the `Commit` phase, the LAD controller sends commit messages carrying the completed DTX's identifier to all MCs. MCs acknowledge the commit message's reception and drain the DTX's corresponding updates from their queue back to PM. Finally, the LAD controller notifies the CPU of the DTX's durability as soon as it receives a commit Ack from any MC.

As the MCs are physically distributed across the chip, commit messages reach each of them at different times, making message reception inherently non-atomic: a system crash may occur when only a subset of the MCs has received a DTX's commit message. In this case, if only the MCs that received the commit message before the crash end up committing the speculative data to persistent memory, the DTX's atomic durability is violated, leaving the system in inconsistent state.

We resolve this challenge by leveraging the fact that all speculative updates are guaranteed to be received by the MCs before the `Commit` phase starts. On a crash, all MCs save a snapshot of their queues' state in PM, which is restored upon system reboot. After an

inter-MC communication phase, all MCs reach a consensus regarding the DTXs that committed before the crash, and thus consistent system state can be recovered. If at least one of the MCs receives a commit message for a given DTX before the crash, at reboot time that MC notifies all other MCs to write that DTX's corresponding updates from their queues back to PM. All other MC queue contents corresponding to uncommitted DTXs are discarded.

3.3 Speculative Buffer Overflow

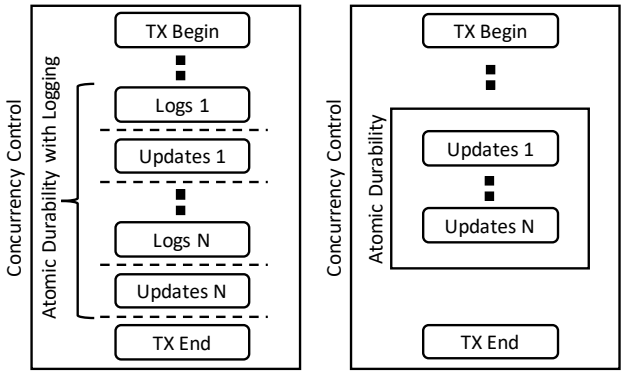
During LAD's `Prepare` phase, all transactional updates have to be buffered within the MCs' persistent queues. For large or multiple concurrent DTXs, these queues may become oversubscribed and overflow. This capacity problem exists in all hardware-based atomicity mechanisms, as the speculative buffer has limited capacity. For example, Intel RTM [16] limits speculative state to the L1 cache and handles overflow by aborting the running transaction, discarding all speculative updates and reverting to a software fallback handler.

LAD handles overflow by falling back to hardware logging, similar to LogTM [29]. When an MC runs out of queue capacity for speculative data, it starts draining its queues by creating a log for the speculative data in a dedicated PM address range, allowing DTXs to proceed and eventually commit. We detail the fallback's implementation in §4.6.

LAD creates undo logs to handle overflow. Undo logs are an appealing design choice for three reasons. First, as crashes are rare, the in-place updates are—most likely—useful work. Second, in-place updates simplify software, as they remove the need for future memory accesses to always check the redo log in PM for potential updates. If a DTX commits and its updates have been partially logged because of an overflow, the DTX's log is simply discarded. The logged updates have already been written in-place in PM, so any updates remaining in the MC queues are written to PM as well, completing the DTX's set of updates. Finally, undo logging also results in a more memory-friendly pattern, as it requires fewer row buffer activations. The old value to be logged is read and replaced with the updated one with a single row activation; writing the undo log entry requires a second row activation. In contrast, redo logging involves a total of three row activations in the general case: write the log entry for the new value (1st) and later read the log entry again (2nd) to apply the required in-place update (3rd).

3.4 Failure and Recovery

LAD always recovers a consistent system state upon reboot by following three sequential steps. First, all MCs exchange information to reach consensus on whether each of the DTXs that were active at the time of failure achieved committed status before the crash. As long as at least one MC received a commit notification for a given DTX, that DTX is deemed committed. Second, if an undo log is present in persistent memory, it is read in chronological order (from oldest to newest entry). For each entry belonging to an uncommitted DTX, the entry's contents are used to restore the corresponding location's value in persistent memory. In the third and last step, the MC goes through the entries of its writeback queues from oldest to newest. The MC writes an entry's value to its corresponding location in persistent memory, if the entry belongs to a committed DTX, or discards it otherwise. §4.8 details the precise steps taken by LAD recovery mechanism's implementation.



(a) Software logging. (b) Atomically durable sections.

Figure 3: Structure of crash-consistent ACID transactions.

3.5 Programmability

From a programming perspective, LAD exposes the simple software interface of an atomically durable section:

```
persistent{<unmodified volatile code>}
```

The interface's simplicity facilitates programming of crash-consistent software and can either be used for DTXs with standalone atomic durability, or can be combined with concurrency control mechanisms to construct ACID transactions.

Fig. 3 graphically illustrates the high-level structure of a crash-consistent ACID transaction with the use of logging (Fig. 3a) versus annotated atomically durable sections (Fig. 3b). The annotated sections interface facilitates the conversion of code originally written for volatile memory to crash-consistent code. In addition, as Fig. 3a shows, a single transaction might dynamically incur multiple logging actions if not all updates are known in advance. In contrast, annotated sections eschew that complication. An ACID transaction can be expressed by encapsulating the durable block within the concurrency control mechanism's critical section, as shown in Fig. 3b. We elaborate on the implications of concurrency control in §3.6.

The programmability aspect of annotated atomically durable sections versus logging for durability is similar to that of transactional memory versus fine-grained locking for concurrency control [11]. Modern PM-aware libraries [15, 42] ameliorate the complexity—and associated programmability burden—of fine-grained logging. However, the performance overhead stemming from additional instructions and synchronization events remains.

3.6 Concurrency Control

The choice of concurrency control mechanism is orthogonal to atomic durability, but of high importance, as the requirements for isolation and atomic durability typically come together. We design LAD to be seamlessly combinable with pessimistic concurrency control mechanisms (i.e., locking), which preclude aborts because of race conditions. Assuming correct software, LAD guarantees that data is persisted in memory in correct happens-before order. Correct software requires transaction schedules to be recoverable.

In a recoverable schedule, a transaction may commit only after all transactions it depends on have committed [9]. Consequently, in the context of pessimistic concurrency control, recoverability is achieved when transactions hold locks until after they commit, which guarantees that no transaction B can access updates of a preceding transaction A before transaction A has already committed and thus its updates have been made persistent. In a recoverable schedule with two dependent transactions $A \rightarrow B$, LAD's commit for transaction A is guaranteed to complete (i.e., all of its updates have reached the MCs) before transaction B can even access any of A's updates. This invariant guarantees that the order of persistence observed in memory follows the order imposed by concurrency control. In §4.5 we demonstrate with an example why irrecoverable schedules can lead to inconsistent memory state and how LAD guarantees correct happens-before order in the presence of correct, recoverable schedules.

In principle, LAD can also be coupled with optimistic concurrency control mechanisms. For example, combination of LAD with a transactional memory [11] implementation, hardware or software, would be a natural fit, as the programming primitives for denoting the concurrency and durability requirements (e.g., TX Begin/End and DTX Begin/End—see Fig.3b) could be trivially merged. However, optimistic concurrency control allows for mid-transaction aborts, which require reverting speculative data updates that, in the general case, may be residing throughout the memory hierarchy. Given that undoing speculative updates throughout the memory hierarchy entails significant complications that are beyond LAD's scope of introducing a novel mechanism for atomic durability, our design assumes coupling with pessimistic concurrency control.

Overall, the increased cost of updates when using PM-aware software tips the scale in favor of pessimistic concurrency control mechanisms, as aborts become more expensive relatively to volatile software. With few exceptions (e.g., [19]), it is therefore common for PM-aware systems to focus on pessimistic concurrency control mechanisms [5, 6, 13, 17, 22, 42].

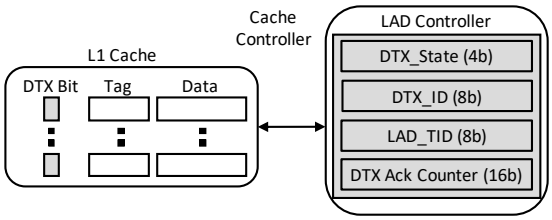
4 LAD IMPLEMENTATION

This section describes our LAD implementation, including ISA and OS extensions, L1D cache and MC modifications, LAD's protocol state machines, fallback mechanism, recovery sequence, and two running examples.

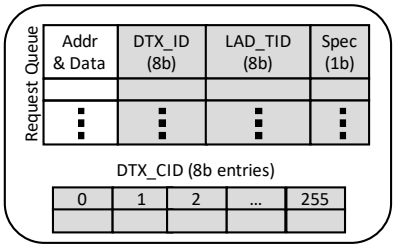
4.1 ISA and OS Extensions

We implement the LAD controller as an extension of each CPU core's L1D cache controller. The core notifies the LAD controller of a DTX's start and end with a pair of instructions: *DTX_Start* and *DTX_End*. These ISA extensions resemble those used in Intel RTM [16] to convey the start and end of a transaction. Because the LAD controller assumes that all stores that arrive to the L1 cache between the reception of a DTX_Start and a DTX_End belong to the running DTX, both instructions wait for the core's store buffer to drain before notifying the LAD controller.

To support thread migrations and context switches, we need to associate a DTX's updates to the software thread executing that DTX. The OS assigns a unique LAD thread ID (henceforth, *LAD_TID)* to a thread that intends to use LAD. The LAD_TID is kept as a new field in the OS kernel's Thread Control Block. Because LAD requires hardware structures to maintain the status of all software threads owning a LAD_TID, we cap the maximum number of threads that can concurrently use LAD to 256. This limitation mainly affects hardware resource provisioning and can be trivially relaxed, if necessary. A DTX's execution can also be disrupted by exceptions or interrupts. In such an event, the LAD controller simply pauses the recording of incoming writes and waits for the DTX to resume. The abstraction of a LAD thread enables resuming the DTX even if the software thread is rescheduled on another core.



(a) L1D cache extensions.



(b) MC extensions.

Figure 4: L1D cache and MC extensions for LAD.

4.2 LAD Controller: L1D Cache Extensions

The LAD controller tracks all of a DTX's updates and coordinates the distributed commit protocol to first transfer these updates to the staging area in the MCs, and then atomically commit all of them to PM. Fig. 4a illustrates the required extensions to the L1 data cache. The LAD controller features a set of DTX tracking structures, comprising five components:

- A *LAD_TID* buffer holds the LAD_TID of the currently running LAD thread.
- A *DTX_State* buffer holds the tracked DTX's state, required by the LAD controller's state machine (see §4.4).
- A *DTX_ID* buffer, incremented for every new DTX. DTX_ID is private per LAD thread; DTX_ID and LAD_TID uniquely identify a DTX's updated cache blocks.
- A *DTX bitvector*, featuring one bit per L1 cache block. A set bit indicates that the corresponding cache block was updated by the DTX that is currently running.
- A *DTX Ack Counter* buffer tracks the number of outstanding messages from the LAD controller (flushes or commits) waiting to be acknowledged by the MCs.

For a 32KB cache, the DTX tracking structures are 70B in total. LAD can also trivially support cores with multiple hardware contexts

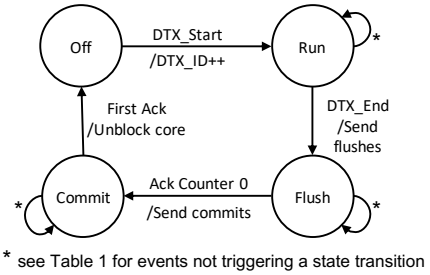


Figure 5: LAD state transition diagram.

	DTX_State	Cache Events			
		Write	Eviction	Coherence Request	Ack
	Off	–	–	–	–
Prepare	Run	Mark	Evict & [*]	[*] & Service	Decrement Ack Counter
Prepare	Flush	–	Evict & [*]	[*] & Service	Decrement Ack Counter
Prepare	Commit	–	–	–	Decrement Ack Counter

[*]: DTX_Flush & Increment Ack Counter

Table 1: LAD controller actions in each state.

(e.g., SMT), by provisioning a copy of the above structures per hardware context.

When the OS scheduler suspends a running LAD thread, the LAD controller clears the DTX bitvector by flushing all marked cache blocks to PM and waits for the DTX Ack Counter to reach zero. DTX_State and DTX_ID are stored in the Thread Control Block along with the thread's LAD_TID, and are restored on the LAD controller of the core the thread gets rescheduled on to enable seamless continuation of the pending DTX.

4.3 Memory Controller Extensions

LAD relies on the existence of persistent request queues in the MCs, and extends them with additional hardware structures for bookkeeping, which are also persistent/battery-backed. Fig. 4b illustrates the added hardware structures. Every entry of the request queue is extended with three additional fields: *DTX_ID*, *LAD_TID*, and a *Speculative bit*. The first two fields identify which DTX_ID and LAD thread each cache block belongs to, while the Speculative bit indicates whether the cache block is a normal writeback request or belongs to an ongoing DTX.

In addition to the added fields in the request queue, the MC maintains a separate direct-mapped structure, the *DTX_CID*, which plays an instrumental role in post-crash recovery, detailed in §4.8. The DTX_CID is a vector of 256 entries, one per LAD thread, which stores each LAD thread's last DTX_ID that was committed. The total added storage at each MC with a standard request queue depth of 64 elements is 392B, corresponding to less than 9% storage overhead per MC. The main impact of increasing the number of threads that can use LAD concurrently is a higher hardware overhead, because of a logarithmic/linear increase in LAD_TID's bitwidth/DTX_CID vector's width, respectively.

4.4 LAD Protocol

We now describe how LAD uses L1 cache and MC hardware additions to deliver atomic durability. The LAD controller located at each core's L1 cache controller drives the protocol's execution. Fig. 5 and Table 1 demonstrate the LAD controller's state transition diagram and the actions taken in each of the states, respectively.

A new DTX starts when a CPU executes a DTX_Start instruction. The CPU stalls until its store buffer has been drained, then DTX_Start retires and the LAD controller is notified of a new DTX's start. The LAD controller increments DTX_ID and transitions to the `Run` state, during which, it marks each written cache block's corresponding bit in the DTX bitvector. These updates are kept as long as possible in the L1 cache to benefit from locality (i.e., coalesce future updates to the same cache block) and are only flushed to the MCs upon a forced eviction by the cache controller, a DTX completion, or an external coherence request.

DTX_End stalls the CPU until its store buffer has been drained, to ensure the LAD controller records all of the DTX's writes. The CPU then notifies the LAD controller, which immediately transitions from the `Run` to the `Flush` state. In the `Flush` state, the LAD controller clears the DTX bitvector, flushing all marked cache blocks from the L1 cache and incrementing its Ack Counter for each flush.

Each DTX_Flush (which carries the cache block's data, DTX_ID, and LAD_TID) is first sent to the LLC where the cache block's value is updated. Then, the DTX_Flush continues to its corresponding MC. This flush operation is the same as `clwb`, and thus does not require modifications to the coherence protocol [40].

The MC stores the cache block carried in DTX_Flush in its persistent request queue—also setting the queue entry's DTX_ID, LAD_TID and Speculative bit fields—and sends an Ack back to the originating LAD controller. For each Ack, the LAD controller decrements its Ack Counter; when Ack Counter = 0, the controller transitions from the `Flush` to the `Commit` state. By the end of the `Flush` state, all of the DTX's updates have been transferred to the MCs' persistent queues, where they are stored in speculative state.

Finally, in the `Commit` state, the LAD controller sends a DTX_Commit message carrying the committing DTX's ⟨LAD_TID, DTX_ID⟩ to all MCs. On reception of DTX_Commit, each MC (i) atomically updates its DTX_CID vector (DTX_CID[LAD_TID] = DTX_ID); (ii) clears the speculative bit of all entries in the request queue belonging to ⟨LAD_TID, DTX_ID⟩; and (iii) sends an Ack back to the message's originating LAD controller. The DTX completes as soon as the first Ack from any MC is received, a policy that is particularly beneficial on multi-socket processors, where the latency to remote sockets' MCs is considerably higher than local ones.

Race conditions. While in the `Run` state, the LAD controller only flushes L1-resident cache blocks updated by the DTX if necessary. An updated cache block may be forced to leave the L1 before the DTX enters the `Flush` state, because of a necessary eviction or an external coherence message (see Table 1). If a coherence message for a marked block arrives from a peer cache, the LAD controller flushes the requested cache block, updating the LLC's contents, and then the LLC responds to the peer cache's request. Theoretically, a race between a DTX_Flush and a subsequent DTX_Flush of the same cache block with an updated value from a peer cache can occur. However, the concurrency control mechanism is expected

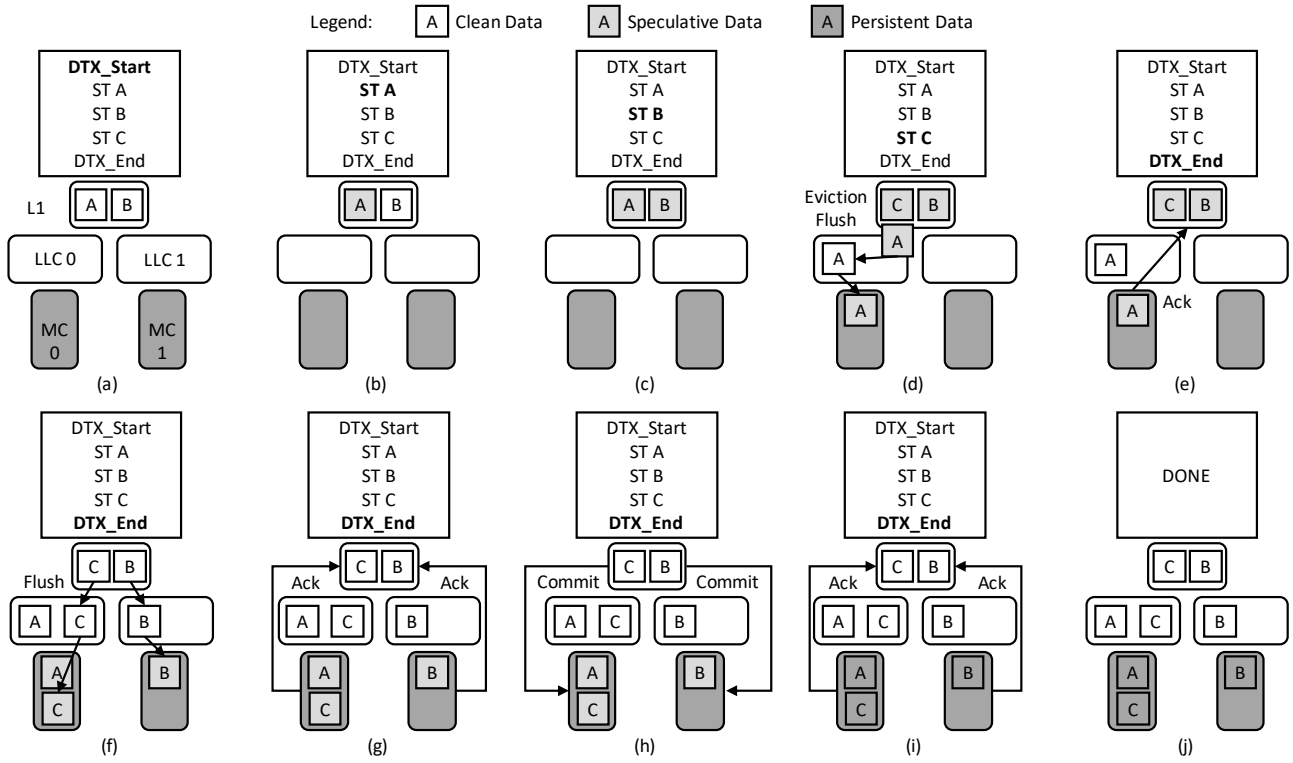


Figure 6: LAD running example.

to enforce proper isolation across racing DTXs, rendering such races—and thus risk of update order inversion—impossible. The `Prepare` phase's synchronous nature guarantees that DTX_Flushes of concurrent DTXs will reach the MCs in the same order as the one dictated by concurrency control.

4.5 Running Examples

Fig. 6 demonstrates LAD's operation with an example. In step (a), the CPU starts executing a DTX (DTX_Start), notifying the LAD controller about the new DTX. The LAD controller transitions to the `Run` state and thus starts marking all of the CPU's writes in the L1. Cache blocks A and B are already present in the L1 cache in a clean state. In steps (b) and (c), the CPU updates A and B and the LAD controller marks their corresponding bits in the DTX bitvector.

In step (d), the CPU writes cache block C, which results in a conflict miss that triggers the eviction of cache block A. As described in §4.4, A is flushed to its corresponding LLC tile and MC. The LAD controller increments its Ack Counter and clears the corresponding bit in the DTX bitvector, which is then set again for the newly arrived cache block C. MC 0 receives the flushed cache block A and stores it in its request queue, marking it as speculative.

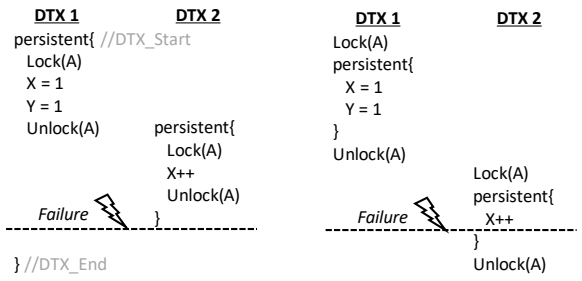
In step (e), the CPU executes DTX_End, notifying the LAD controller to transition to the `Flush` state. Meanwhile, MC 0 sends an Ack for the cache block flushed in step (d), and the LAD controller decrements its DTX Ack Counter.

In step (f), the LAD controller starts flushing all marked cache blocks from the L1 cache, sending them to their corresponding MCs through the LLC. The LAD controller increments its Ack Counter with every flush performed, and clears each flushed block's corresponding bit in the DTX bitvector. The MCs receive the flushed blocks and store them in their request queues, marked as speculative. The CPU stalls on DTX_End waiting for a signal of durability completion from the LAD controller.

In step (g), each MC independently acknowledges each flush received. For every received Ack, the LAD controller decrements its Ack Counter. Once the counter reaches zero, the LAD controller transitions to the `Commit` state, which also marks the transition from the `Prepare` to the `Commit` phase.

In step (h), the LAD controller sends out commit messages to all MCs, which asynchronously reach the MCs. On receiving a commit message, each MC clears the speculative bit of all cache blocks belonging to the committing DTX—this is the point the DTX becomes effectively persistent. Then, in step (i) each MC acknowledges the commit request. As soon as the LAD controller receives the first Ack from an MC, it transitions to the `Off` state and sends a notification to the CPU (step (j)). The DTX_End instruction retires, unblocking the CPU. At this time, the DTX has been made atomically durable in a logless and distributed manner.

As detailed in §3.6, LAD preserves the correct happens-before order of updates to memory under correct use of concurrency control (i.e., recoverable transaction schedules). Fig. 7's example demonstrates why irrecoverable schedules can lead to corrupted persistent memory state (Fig. 7a) and why recoverable schedules (Fig. 7b) guarantee that the ordering of memory updates is preserved.



(a) Irrecoverable schedule. **(b) Recoverable schedule.**

Figure 7: Interplay of LAD's atomically durable sections with concurrency control. Assume initial values $X = Y = 0$.

In Fig. 7a's irrecoverable schedule, transactions exit their critical section before committing (i.e., LAD's atomically durable section is not encapsulated within the critical section, as required). At the time of failure, DTX 2 has committed, so $X = 2$ is made persistent in memory. As the failure occurred during DTX 1's `Prepare` phase, DTX 1's updates to X and Y may or may not have reached the MCs. Regardless, all of DTX 1's updates will be discarded by the LAD recovery protocol upon reboot. As a result, the memory contents after recovery will be $(X, Y) = (2, 0)$, which is an inconsistent state. Note that the culprit is erroneous use of concurrency control.

Fig. 7b shows the same sequence of DTX execution, where LAD's atomic persistence block is correctly placed within the critical section. At the time of failure, DTX 2's updated value of X may or may not have reached the MCs. However, as DTX 2 didn't complete its commit protocol, any DTX 2 updates received by the MCs will be discarded at the time of recovery, bringing memory to a consistent state ($(X, Y) = (1, 1)$). Fig. 7b's example demonstrates why LAD always preserves the correct order of updates, when properly combined with concurrency control that only allows recoverable schedules. The invariant is that no DTX 2 can access (read or write) a value updated by a preceding DTX 1, before all of DTX 1's updates have reached the MCs.

4.6 Fallback Logging

Because MC queues are bounded, overflow with speculative writebacks is possible. As described in §3.3, LAD's fallback in case of such overflow is that the MC starts draining its request queue in PM in chronological FIFO order, implementing undo logging. Dedicated space is pre-allocated by the OS for this purpose on a per-MC basis, and after allocation it is directly managed by the MCs. In our implementation, logging starts when 80% of the request queue's entries are occupied by speculative cache blocks.

Each undo log entry contains the logged cache block's old value and ⟨LAD_TID, DTX_ID⟩. Before a speculative cache block is logged, its corresponding old value is read from PM and the new value is written back to PM. The retrieved old value is used to create the undo log entry. A background OS thread periodically reclaims the memory used by the undo logs, and also allocates additional logging space for the MCs if necessary. In the rare case the MCs near depletion of their logging space, an interrupt is raised to notify the OS thread to allocate additional space, a practice similar to prior work managing logging in hardware [20, 46].

4.7 Interplay with Coherence

LAD does not require any coherence protocol modifications. This section serves as an informal proof that the LAD protocol's interplay with coherence is deadlock free. We briefly go over all resources that are acquired and freed in the course of LAD's operation, in the L1 cache, the LLC, the MCs and the on-chip interconnect.

L1 cache: LAD tracks blocks in the cache but releases them immediately upon a coherence request. Only the Ack counter—a LAD-specific resource independent of coherence—remains as allocated state waiting for the pending DTX_Flushes to be acknowledged.

LLC: LAD does not reserve any resources at the LLC. DTX_-Flushes only update the cache block's value in the LLC.

MC: LAD reserves entries in the MC queues while a DTX is advancing and more cache blocks are flushed. While these MC queues have limited capacity, when under pressure, their entries are drained to PM as logs. With PM assumed to be an "infinite" resource for LAD's purposes (i.e., there is enough space allocated for logging at all times), LAD's state can always be drained to free up the MC queues.

Interconnect: Assuming an interconnect with different VCs for coherence requests and replies, we leverage these same VCs to separate LAD messages into requests (DTX_Flush, DTX_Commit) and replies (Acks). All LAD messages can always be processed to completion upon reception at the L1 cache, LLC, or MC. Thus, LAD messages cannot block coherence messages in the interconnect.

4.8 Failure and Recovery

LAD assumes that in case of a crash, the contents of the MC queues are preserved. To achieve that, we leverage technology that already exists in persistence-aware MCs of Intel servers [36, 39], which are sufficiently battery backed to flush their queues' contents into memory upon failure. While our approach is similar, LAD's flush-on-failure policy requires slight modifications to the current policy implemented in MCs, because MC structures in LAD hold both committed and speculative state. Therefore, instead of draining all pending requests in the queue to their corresponding locations in PM upon failure, our modified MC differentiates between writebacks marked by LAD and normal ones. LAD writebacks are drained in chronological order into "purgatory", a dedicated space of a few KBs in PM, where they are stored as temporary—yet non-volatile—state to be recovered after reboot. All other pending writebacks are normally written back into their corresponding locations in PM. Upon reboot, the pre-crash contents of the MCs' queues can be recovered by reading back the purgatory's contents.

The recovery process should be performed as soon as the system restarts, before the OS resumes normal operation. Consistent memory state is established by following three steps:

- **Step 1:** The DTX_CID vector $V_i = \langle c_{i1}, ..., c_{iN} \rangle$ (where N = # LAD threads—256 in our implementation) of every MC_i is read. V_i contains the DTX_ID of the last committed DTX per LAD thread. A single vector

$$V_{commit} = \langle \max_{i=1}^{M}(c_{i1}), ..., \max_{i=1}^{M}(c_{iN}) \rangle, \; M = \#\ of\ MCs$$

Name	Writes / DTX (cache blocks)	Description
TATP	1	Update location transaction in TATP
RBT	2–10	Ins./Del. entries in a Red-Black tree
CQ	4	Insert/Delete entries in a queue
PC	8	Modify values in a hash-table
SPS	16	Random swaps of array elements
TPCC	10–35	New Order transaction in TPC-C

Table 2: Evaluated benchmarks.

is compiled and sent to all MCs. The V_{commit} vector summarizes all the MCs' consensus regarding each core's last committed DTX. Doing so covers the case where power failure resulted in partial reception of a DTX_Commit message marking a DTX's end.

- **Step 2:** Each MC goes through its private undo log, from oldest to newest entry. For each entry ⟨LAD_TID, DTX_ID⟩ with DTX_ID > V_{commit}[LAD_TID], the MCs restore the corresponding location's value in PM with the old value found in the undo log.
- **Step 3:** The contents of the purgatory are read. Entries belonging to committed DTXs are written directly to their corresponding location in PM in chronological order, while entries belonging to uncommitted ones are discarded. After all three steps are done, the undo log and purgatory are cleared and the system is ready to resume normal operation.

The above steps assume coordinating state machines, one per MC. Alternatively, a single core in BIOS mode (e.g., as part of the memory count-up test) can perform them sequentially for each MC. If the system undergoes another power failure during recovery, the same recovery process restarts anew, as all recovery-related operations are idempotent and the state of the undo log and purgatory are preserved until recovery completes successfully. Thus, LAD's recovery procedure can trivially sustain recursive failures.

5 METHODOLOGY

Applications. We use a benchmark suite for durable transactions, taken from prior work [10, 23]. Table 2 briefly describes each benchmark and lists the number of updated cache blocks per DTX in each case. The datasets of all the benchmarks exceed the capacity of on-chip caches but are memory resident, except for RBT that operates on an LLC-resident dataset. We focus on small transactions, which—in the common case—can leverage LAD to complete in a logless fashion. As demonstrated in prior work, persistence-aware software is dominated by such small transactions [30].

TATP and CQ have small transactions with a very small read/write ratio. RBT also has small transactions, but with more reads as it traverses the tree for each operation. As the trees are small and fit in the private cache, volatile transactions do not generate read/write traffic to memory. PC, SPS, and TPCC have large transactions with increasing read intensity. All the workloads execute back to back transactions in order to stress LAD. Therefore, they represent an extreme case as real-world applications are expected to execute other operations beside durable transactions.

System architecture. We assume a tiled 16-core CPU with a distributed LLC. The memory address space is horizontally partitioned between volatile and persistent regions. We model PM as battery-backed DDR4 DRAM, a representative form of PM deployment in modern commercial datacenters [21]. We pin our workloads on 15 cores, leaving one core for the OS. To stress LAD's distributed commit protocol, we model four MCs. While our main evaluation is based on a single-socket setup, we also evaluate the impact of increased latency to reach a subset of the MCs, which typically occurs when scaling beyond a single socket. For these experiments, we emulate a dual-socket effect with half of the MCs injecting an additional delay of 50ns in their responses.

Cores	16× ARM Cortex-A57 64-bit, 2GHz, OoO, TSO 3-wide dispatch/retirement, 128-entry ROB
L1 Caches	32KB 2-way L1D, 48KB 3-way L1I 64-byte blocks, 2 ports, 32 MSHRs 2-cycle latency (tag+data)
LLC	Shared block-interleaved NUCA, non-inclusive 8MB, 16-way, 1 bank/tile, 6-cycle access
Coherence	Directory-based MESI
Interconnect	2D mesh, 16B links, 3 cycles/hop
DRAM Timing	t_{CK} = 0.625 ns, t_{RAS} = 24 ns, t_{RCD} = 13.75 ns t_{CAS} = 11.2 ns, t_{WR} = 10 ns, t_{RP} = 13.75 ns
LAD	8 sets of DTX Tracking Structures per L1 cache 4 MCs (2 with 50ns socket overhead) 64-entry request queue per MC [23]

Table 3: System parameters for simulation on Flexus.

We compare LAD against the most relevant state-of-the-art hardware design for logless atomic durability, Kiln [46]. However, as the original Kiln design does not consider a distributed NUCA LLC, we extend its functionality by adding LAD's distributed commit protocol to the LLC controllers rather than the MCs. We optimistically model a persistent battery-backed LLC rather than STT-RAM, thus enabling persistence without any latency penalty compared to a volatile SRAM LLC. This configuration, *LAD-LLC*, represents an idealized implementation of a logless atomic durability mechanism.

Evaluated configurations. We evaluate four configurations:

(1) *Volatile*: Transactions touch only volatile data. This configuration represents ideal performance, as there is no cost associated with providing atomic durability.
(2) *LAD-LLC*: Transactions attain durability by writing updated data to the persistent (battery-backed) LLC.
(3) *LAD*: Transactions attain durability by writing updated data to persistent (battery-backed) MCs. By default, a DTX commits after the LAD controller receives the first Ack from an MC. We also evaluate a *LAD-Base* configuration that lacks this optimized commit-after-first-Ack policy (i.e., the LAD controller waits for all Acks).
(4) *SW Logging*: Transactions use software logging for atomic durability, and undo logs are written to persistent MCs using `clwb` operations.

Simulation. We evaluate all configurations using Flexus [44], a full-system cycle-accurate simulator coupled with DRAMSim2 [35] for DRAM simulation. Table 3 summarizes simulation parameters.

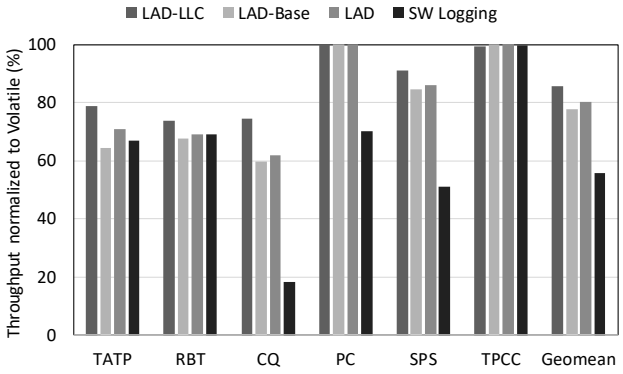


Figure 8: Performance with synchronous transactions.

6 EVALUATION

6.1 Atomic Durability Performance

Fig. 8 shows the throughput (transactions per second) of different configurations normalized to Volatile. Volatile outperforms LAD-LLC by 14% on average, as in LAD-LLC every transaction blocks the CPU until all of the transaction's updated L1-resident data are flushed to the persistent LLC because of the software's strict synchronous durability requirements. Workloads are affected differently by the atomic durability requirement, depending on their memory access patterns. In general, the durability property for workloads with small transactions is costlier, because of the higher frequency of CPU-blocking events.

Software logging severely degrades the throughput of most workloads, by about 45% on average. Only TPCC remains unaffected by the overhead of logging, which is amortized by TPCC's large read-heavy transactions. On the other end of the spectrum, CQ is the most write-intensive workload with high MLP and tiny transactions. The multiple serialization points introduced by logging severely hurt its performance. LAD outperforms software logging across the board because it reduces the instruction footprint and results in fewer writes to memory, as we will demonstrate in §6.3.

LAD also closely follows the performance delivered by the optimistic LAD-LLC configuration. The small performance gap is attributed to LAD's requirement for flushes to reach the MCs rather than the LLC. Despite that, and despite the two orders of magnitude smaller capacity for speculative buffering of transaction updates (16KB vs. 8MB), LAD performs within 5% of LAD-LLC on average. Overall, our results indicate that the cost and effort of battery backing the LLC only yields minuscule performance improvements for crash-consistent software as compared to LAD, which only requires persistent MCs. The latter is readily available in modern server CPUs, indicating our design's practicality.

To better understand the performance impact of our distributed commit protocol, Fig. 8 also includes a version of LAD where a DTX is committed only after Acks are received from every MC in the `Commit` phase (LAD-Base). On our single-socket experiments, the modest performance degradation (~2% avg) of LAD-Base compared to LAD demonstrates two expected outcomes. First, our protocol spends more time in the `Flush` state than the `Commit` state, because `Flush` involves moving data from the L1D cache to the MC rather than a simple round-trip of a control message. Second, the minimal performance hit taken when waiting for the last Ack in the `Commit` state is a direct consequence of the single-socket system, where the cores' distance to all MCs is almost uniform. Despite the small average performance difference between LAD and LAD-Base, workloads with very short DTXs still benefit from skipping the wait for the last Ack. For example, TATP spends similar time in the `Flush` and `Commit` state; as a result, LAD is 7% faster than LAD-Base.

	Single-Socket (Fig. 8)	Dual-Socket
LAD-LLC	86%	84%
LAD-Base	78%	72%
LAD	80%	77%
SW Logging	56%	48%

Table 4: Throughput (geomean) of single-socket and dual-socket configurations normalized to Volatile.

Table 4 summarizes the average throughput of all aforementioned configurations on a single-socket system and compares them to a dual-socket system. The performance gap of all configurations compared to Volatile grows, because of the dual-socket's increased latency effect on the synchronous operations introduced by the mechanisms enabling atomic durability. For the same reason, waiting for only one Ack by the MCs at the `Commit` phase now becomes more significant, increasing the performance difference between LAD and LAD-Base to ~5% on average and to 12% for TATP.

In summary, LAD enables synchronous atomically durable transactions at low cost, being within 20% of the volatile single-socket system's performance. This number is close to what Kiln [46]—a state-of-the-art mechanism for synchronous atomic durability closely emulated by our LAD-LLC configuration—delivers, but with significantly less intrusive hardware changes. LAD only requires extending the L1D caches and MCs with additional state stored in volatile SRAM buffers, rather than making the entire LLC non-volatile. Finally, LAD's performance is not directly comparable to work proposing asynchronous durability (e.g., DudeTM [25]), which relaxes the strict semantics of synchronous atomic durability for performance gains.

6.2 LAD Overhead Breakdown

Table 5 shows each workload's per-transaction latency for LAD, including the time spent in the `Prepare` and `Commit` phase. Unsurprisingly, the `Prepare` phase's overhead is always higher than the `Commit` phase's, as it involves flushing data from the L1 cache to the MCs. As the `Commit` phase involves a roundtrip of control messages between the L1 and the MCs, its latency should be largely insensitive to the workload. The small variability observed in the `Commit` phase's latency is attributed to the workload-dependent contention in the interconnect. We verify this hypothesis on an unloaded system by using single-threaded instances of the workloads, where we indeed observe a fixed `Commit` phase latency of 30 cycles. Consequently, there is little room for performance improvement by accelerating the `Commit` phase.

Proactive flush mechanisms [18, 21] have the potential of reducing the `Prepare` phase's overhead, by removing all but a single cache block's flush from the critical path. We observe that even an ideal proactive flush mechanism, with 100% accuracy in flushing cache blocks right after their last update within a DTX, would yield

Workload	Total	`Prepare` phase	`Commit` phase	Perf. improv. upper bound
TATP	491	108	42	4%
RBT	269	86	35	1%
CQ	771	117	48	3%
PC	1614	158	48	4%
SPS	3062	165	45	3%
TPCC	17019	91	36	0%

Table 5: LAD per-transaction latency breakdown (in cycles) and upper performance improvement bound for an ideal early flush mechanism.

marginal performance improvements. We estimate an upper improvement bound by replacing each workload's measured `Prepare` phase latency with the absolute minimum latency required to flush one block from the L1 to the MC (L1→LLC→MC→returning Ack). By setting that latency to 80 cycles (equal to TATP's `Prepare` phase latency on an unloaded system; TATP always has a single cache block flush on the critical path), the estimated performance improvement for each workload is displayed in Table 5's last column.

These results indicate that the `Prepare` phase's overhead is not bandwidth-, but latency-bound. Drastically reducing the remaining performance margin between LAD and Volatile requires exploring asynchronous commit mechanisms, which may require relaxation of the strict semantics of synchronous atomicity.

6.3 Sensitivity to MC Request Queue Size

Fig. 9 shows the memory accesses per transaction, averaged across all workloads, for our three evaluated crash-consistent configurations with varying MC queue sizes. The secondary y-axis shows performance normalized to Volatile with the same MC queue size. Software logging increases the number of memory accesses by more than 3×, mainly because of writing log entries and flushing in-place updates. Reads also increase, as the log's memory locations are first read by the CPU. LAD only increases memory accesses by the number of cache blocks written per transaction, because they have to be written back to memory to attain persistence. LAD-LLC's memory access pattern is the same as the Volatile baseline's, as flushes for durability purposes are absorbed by the LLC and do not reach the memory.

We now focus on how MC queue size affects the frequency of LAD's fallback logging mechanism and, conversely, performance. A commonly modeled MC queue size is 64 entries. For our evaluated benchmark suite and a 64-entry MC queue, LAD's fallback mechanism is never triggered. We only observe measurable memory traffic caused by the fallback mechanism when we shrink the MC queue eight-fold, to 8 entries, which results in < 3% performance drop. The worst cases we observe for a queue size of 8 are for TPCC and CQ, when the resulting fallback logging introduces ∼ 12% more memory writebacks as compared to the baseline queue size of 64. These additional writebacks hurt TPCC's and CQ's performance by <1% and 9%, respectively. CQ is affected observably more by the additional writes, because of its bandwidth-intensive nature. Finally, software logging suffers significantly more from smaller MC queues, because of the higher pressure it puts on the memory. Our results validate that, in the common case, LAD delivers atomic durability in a logless manner, despite the limited buffering space available at the MCs.

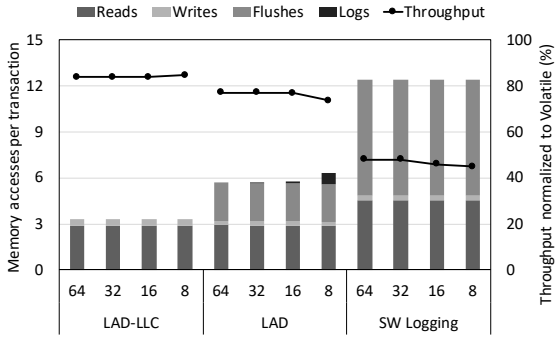


Figure 9: Sensitivity to memory controller queue size.

6.4 Impact of Non-Volatile Memory

While the main body of our evaluation is based on modeling PM after battery-backed DRAM, the same observations hold for non-volatile memory (NVM) technologies. NVM has a broad spectrum of access latency and read/write disparity; however, NVM used as DRAM replacement will most likely fall in the faster end of that spectrum. We therefore repeated our evaluation using 60/150ns read/write memory access latency, representative of a fast NVM.

Despite changes in absolute numbers, relative performance across configurations marginally shifts as compared to the results presented in §6.1 and therefore we omit detailed data for brevity. Our takeaways regarding fallback logging frequency (§6.3) remain unaffected as well. Our MC queue sizing remains sufficient for NVM, because our workloads are not bottlenecked by memory bandwidth, and, as discussed in §6.3, LAD's 64-entry MC queue is 4–8× overprovisioned for battery-backed DRAM.

7 CONCLUSION

We presented LAD, a logless atomic durability mechanism that utilizes data multiversioning inherent in memory hierarchies. We leverage persistent MCs, which are already featured in modern servers, to accumulate speculative state and commit atomically to PM. LAD features a distributed protocol in hardware to manage the speculative state residing across the chip's MCs and make atomic commit decisions. Our design enables atomically durable synchronous transactions at a performance cost as low as 20%, making intrusive hardware modifications to extend the persistent domain from the MCs further up in the memory hierarchy hardly justifiable.

ACKNOWLEDGMENTS

We are grateful to Aasheesh Kolli and Vaibhav Gogte for giving us access to their PM workloads, Anirudh Badam, Jim Larus, Paolo Ienne and Dmitrii Ustiugov for insightful comments and suggestions in early versions of this work. We also thank Mario Drumond, Mark Sutherland, Arash Pourhabibi, Zilu Tian, Yunho Oh, Ognjen Glamocanin, Rishabh Iyer and Marina Vemmou for their feedback and support. This work was partially supported by Huawei Technologies as part of the "Rack-Scale In-Memory Computing" project, YB2016020020.

REFERENCES

[1] 2018. Intel Mesh Interconnect Architecture. https://en.wikichip.org/wiki/intel/mesh_interconnect_architecture.

[2] 2019. AMD Zen Microarchitectures. https://en.wikichip.org/wiki/amd/microarchitectures/zen#Multiprocessors.

[3] Rajeev Balasubramonian, Norman P. Jouppi, and Naveen Muralimanohar. 2011. *Multi-Core Cache Hierarchies.* Morgan & Claypool Publishers.

[4] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. 2007. BulkSC: bulk enforcement of sequential consistency. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*. 278–289.

[5] Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. 2014. Atlas: leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 433–452.

[6] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)*. 105–118.

[7] Nachshon Cohen, Michal Friedman, and James R. Larus. 2017. Efficient logging in non-volatile memory by exploiting coherency protocols. *PACMPL* 1, OOPSLA (2017), 67:1–67:24.

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. 133–146.

[9] Ramez Elmasri and Shamkant B. Navathe. 2010. *Fundamentals of Database Systems, 6th Edition.* Addison-Wesley.

[10] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for synchronization-free regions. In *Proceedings of the ACM SIGPLAN 2018 Conference on Programming Language Design and Implementation (PLDI)*. 46–61.

[11] Tim Harris, James R. Larus, and Ravi Rajwar. 2010. *Transactional Memory, 2nd edition.* Morgan & Claypool Publishers.

[12] Joel Hruska. 2018. Intel Optane DC. https://www.extremetech.com/extreme/270270-intel-announces-new-optane-dc-persistent-memory.

[13] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the 2017 EuroSys Conference*. 468–482.

[14] Intel Corporation. 2019. Intel ISA Manual. https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf.

[15] Intel Corporation. 2019. Intel PMDK. http://pmem.io/pmdk/.

[16] Intel Corporation. 2019. Intel RTM. https://software.intel.com/en-us/node/524025.

[17] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXI)*. 427–442.

[18] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient persist barriers for multicores. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 660–671.

[19] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*. 452–465.

[20] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 361–372.

[21] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. 2017. Viyojit: Decoupling Battery and DRAM Capacities for Battery-Backed DRAM. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 613–626.

[22] Aasheesh Kolli, Steven Pelley, Ali G. Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXI)*. 399–411.

[23] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali G. Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated persist ordering. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 58:1–58:13.

[24] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*. 2–13.

[25] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*. 329–343.

[26] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *Proceedings of the 32nd International IEEE Conference on Computer Design (ICCD)*. 216–223.

[27] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the 2017 EuroSys Conference*. 499–512.

[28] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162.

[29] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. 2006. LogTM: log-based transactional memory. In *Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 254–265.

[30] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*. 135–148.

[31] Matheus Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *Proceedings of the 24th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 336–349.

[32] Xuehai Qian, Wonsun Ahn, and Josep Torrellas. 2010. ScalableBulk: Scalable Cache Coherence for Atomic Blocks in a Lazy Environment. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 447–458.

[33] Xuehai Qian, Josep Torrellas, Benjamin Sahelices, and Depei Qian. 2013. BulkCommit: scalable and fast commit of atomic blocks in a lazy multiprocessor environment. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 371–382.

[34] Jinglei Ren, Jishen Zhao, Samira Manabi Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 672–685.

[35] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters* 10, 1 (2011), 16–19.

[36] Andy Rudoff. 2016. Deprecating the PCOMMIT Instruction. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[37] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: a flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 178–190.

[38] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*. 175–186.

[39] SNIA. 2014. NVDIMM Messaging and FAQ. www.snia.org/sites/default/files/NVDIMM%20Messaging%20and%20FAQ%20Jan%2020143.pdf.

[40] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool Publishers.

[41] Andrew S. Tanenbaum and Maarten van Steen. 2007. *Distributed systems - principles and paradigms, 2nd Edition.* Pearson Education.

[42] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)*. 91–104.

[43] Hu Wan, Youyou Lu, Yuanchao Xu, and Jiwu Shu. 2016. Empirical study of redo and undo logging in persistent memory. In *Proceedings of the 5th IEEE Non-Volatile Memory Systems and Applications Symposium*. 1–6.

[44] Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. 2006. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro* 26, 4 (2006), 18–31.

[45] Andrew W. Wilson Jr. and Richard P. LaRowe Jr. 1992. Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture. *J. Parallel Distrib. Comput.* 15, 4 (1992), 351–367.

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