Unleashing the Full Potential of Persistent Memory with Logless Atomic Durability

Siddharth Gupta Alexandros Daglis[†] Babak Falsafi

EcoCloud, EPFL [†]Georgia Institute of Technology

Abstract

Persistent Memory (PM) devices, like 3D-Xpoint, are now commercially available. Software can leverage the combination of its fast access and persistence for significant performance gains. A key challenge for PM-aware software is to maintain high performance while achieving atomic durability requirements. The latter traditionally 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 atomically committing them to PM. LAD employs an on-chip distributed commit protocol in hardware to manage the distributed speculative state each transaction accumulates across multiple MCs. LAD relies on modest hardware modifications to provide atomically durable transactions, delivering up to 80% of ideal—i.e., PM-oblivious software's—performance.

1. Motivation

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. To leverage this capability, software has to follow certain PM-specific contracts to be crash-consistent, namely ensuring that PM's contents allow the application to recover to a valid operational state after a system crash. A sudden crash amidst a set of logically atomic updates may result in only a subset of them having reached PM, leading to an inconsistent state. To handle such situations, crash consistency requires special mechanisms that enable reverting partial updates, thereby restoring the data in PM to a consistent state.

Software expresses crash consistency contracts using high-level programming abstractions, such as that of an atomic transaction: a set of writes that either take place atomically, or not at all. A transaction defines a unit of consistent state change in the software, as it takes the system state from one consistent state to another via arbitrary sets of writes in between. In the context of PM, either all of a transaction's writes have to be made durable, or all are collectively discarded in case of a crash. In other words, a transaction's set of writes requires *atomic durability.*

Appeared in MICRO'19 as *Distributed Logless Atomic Durability with Persistent Memory.* URL: https://infoscience.epfl.ch/record/270293?ln=en

Guaranteeing atomic durability requires two invariants to hold before any in-place PM updates can be performed:

- *Inv1.* Before committing the transaction, the complete version of both old and new data must co-exist.
- *Inv2.* Both versions must be present in persistent memory, so that they can survive an unexpected crash.

If both invariants are satisfied, successful transition to a consistent state is guaranteed. If the new version of updates is incomplete, the state is reverted back to the old version. Otherwise, the new version becomes the final state and the old version is discarded.

Traditionally, atomic durability is achieved via write-ahead logging: an explicit log containing the old data values is created (*Inv1*) and written in PM (*Inv2*) before any in-place updates are applied in PM. If a crash happens during this process, the log can be used to recover the data back to its old state. Otherwise, the log is discarded to indicate transaction completion. While effective, logging comes with performance overheads because of additional instructions per transaction and PM write ordering requirements between logs and in-place data updates. In addition, the extra PM writes can shorten the PM device's lifespan.

We observe that we can leverage basic building blocks that already exist in modern CPUs to meet the two invariants necessary for atomic durability, without explicit logging and its inherent overheads. First, memory hierarchies naturally offer data multiversioning; with additional control on update propagation to lower levels of the hierarchy, we can achieve *Inv1.* Second, modern servers that support PM already feature persistent memory controllers (MCs). We can leverage the limited but persistent buffering space in them to achieve *Inv2.* We thus introduce LAD, a mechanism that requires a limited set of hardware extensions to the CPU and MCs to achieve atomic durability in a *logless* fashion.

2. Logless Atomic Durability

Fig. 1a shows LAD's high-level idea. Modern MCs have persistent queues, meaning their contents are not lost in case of power failure. We extend these queues with the ability to mark specific entries as speculative. As long as an entry remains speculative, it is not drained back to PM.

Fig. 1b shows LAD's two phases of transaction handling. The core initially reads the transaction's target values from PM into its private volatile L1 cache, where they are updated and marked as speculative. If a speculative value is evicted from the L1, it is flushed to the MC, where it remains marked as speculative. Once the core completes all of the transaction's updates, it explicitly flushes all remaining speculative values from its L1 to the MCs, waiting for each flush to be acknowledged.

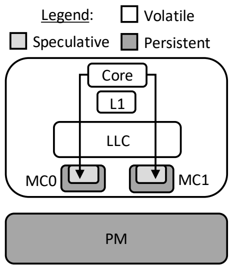
Once all updates are acknowledged, LAD enters its protocol's second phase, where the CPU sends a commit message to the MCs. A commit message atomically turns all the speculative blocks in the MC queues to non-speculative, which can now be drained to PM. The core can continue normal execution the moment MCs acknowledge reception of the commit message, while the data drain to PM can happen asynchronously without data loss risk, because MC queues are persistent. The transaction's atomic persistence is thus guaranteed without explicit log creation.

Distributed speculative state management. Modern CPUs feature multiple MCs, thus, under LAD's protocol, each running transaction accumulates speculative state that is physically distributed. As a result, controlling a single transaction's status requires the core to communicate with multiple MCs. With each MC being an independent entity, this problem is equivalent to an atomic commit problem in a distributed system. Therefore, LAD employs a variant of the well-known two-phase commit (2PC) protocol.

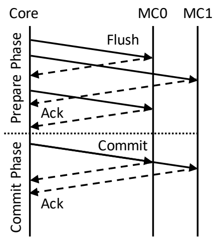
However, using a 2PC-like protocol alone does not guarantee atomic durability in the presence of crashes. At all times, all MCs need to have a consistent view of every transaction's success or failure and act accordingly: drain the transaction's buffered updates to PM or discard them. Inter-MC consensus is not trivially guaranteed, because commit message propagation on the on-chip interconnect is not atomic. In case of a crash, it is possible that only a subset of MCs receives a given transaction's commit message.

LAD solves this complication with a recovery protocol triggered upon post-crash system reboot, during which the MCs communicate with each other to determine the latest committed transactions. After consensus is reached, all MCs perform the same action for every transaction that had speculative updates buffered in MC queues at the time of the crash. Speculative entries belonging to committed transactions are written back to their corresponding location in PM; the rest are discarded. Thus, atomic durability is guaranteed.

Additional key design considerations. While this summary presents LAD's main idea, it omits a range of important design aspects that are covered in the full paper: (i) MC buffer overflow handling; (ii) required hardware extensions; (iii) recovery sequence; (iv) transaction concurrency.



(a) System architecture.



(b) Protocol timeline.

Figure 1: LAD overview and distributed commit protocol.

3. Performance Benefits and Significance

LAD drastically ameliorates the overheads of atomic durability, achieving 80% of volatile software's performance. We also investigate the effect of raising the persistence boundary from the MCs to the LLC, an approach similar to Kiln[1], and find that such an aggressive approach only outperforms LAD by 5%, despite the latency reduction and 500$\times$ buffering capacity increase. Thus, introducing persistent LLCs not only involves technological challenges generally considered to be- impractical, but also yields diminishing returns. LAD demonstrates that persistent MCs are a *sufficient* building block for a hardware mechanism that provides atomic durability to replace software logging and its associated high performance overheads. Overall, LAD's key design choice of leveraging persistent MCs is both well-performing and practical, and thus has the potential to influence the design of PM-equipped systems and reach commercialization.

From a broader perspective, LAD suggests that CPU-centric atomic durability mechanisms (e.g., logging), face a fundamental roadblock: the semantic gap between the CPU's requirements and the interface exposed to it by the memory hierarchy. The abstraction of a logically atomic set of updates is only understood by the CPU, while persistence is a quality offered and controlled by memory (and by extension, the MCs). However, because of the wide gap between the CPU and memory in modern systems—taking the form of a deep volatile cache hierarchy—the memory interface exposed to the CPU does not offer any persistence guarantees.

Lacking the means to directly convey high-level semantic information to the MCs, the CPU has to resort to the scant semantics exposed by the memory hierarchy to compose the application requirement of atomic durability via metadata creation and update ordering. Addressing this inefficiency requires bridging the semantic gap with a synergistic mechanism that combines the CPU's high-level semantic information with the MC's low-level persistence quality. LAD demonstrates the potential of enabling such synergy, by promoting the MC from a stateless entity that handles each received cache block independently to a control agent that directly interacts with the CPU via a control plane.

[1] J. Zhao et al. *Kiln: closing the performance gap between systems with and without persistence support.* In MICRO 2013.