ASAP: Architecture Support for Asynchronous Persistence

Ahmed Abulila*
University of Illinois at Urbana-Champaign

Izzat El Hajj
American University of Beirut

Myoungsoo Jung
Korea Advanced Institute of Science and Technology

Nam Sung Kim
University of Illinois at Urbana-Champaign

ABSTRACT

Supporting atomic durability of updates for persistent memories is typically achieved with Write-Ahead Logging (WAL). WAL flushes log entries to persistent memory before making the actual data persistent to ensure that a consistent state can be recovered if a crash occurs. Performing WAL in hardware is attractive because it makes most aspects of log management transparent to software, and it completes log persist operations (LPOs) and data persist operations (DPOs) in the background, overlapping them with the execution of other instructions.

Prior hardware logging solutions commit atomic regions synchronously. That is, once the end of a region is reached, all outstanding persist operations required for the region to commit must complete before instruction execution may proceed. For undo logging, LPOs and DPOs are both performed synchronously to ensure that the region commits synchronously. For redo logging, DPOs can be performed asynchronously, but LPOs are performed synchronously to ensure that the region commits synchronously. In both cases, waiting for synchronous persist operations (LPO or DPO) at the end of an atomic region causes atomic regions to incur high latency.

To tackle this limitation, we propose *ASAP*, a hardware logging solution that allows atomic regions to commit asynchronously. That is, once the end of an atomic region is reached, instruction execution may proceed without waiting for outstanding persist operations to complete. As such, both LPOs and DPOs can be performed asynchronously. The challenge with allowing atomic regions to commit asynchronously is that it can lead to control and data dependence violations in the commit order of the atomic regions, leaving data in an unrecoverable state in case of a crash. To address this issue, *ASAP* tracks and enforces control and data dependencies between atomic regions in hardware to ensure that the regions commit in the proper order.

Our evaluation shows that *ASAP* outperforms the state-of-the-art hardware undo and redo logging techniques by 1.41× and 1.53×, respectively, while achieving 0.96× the ideal performance when no persistence is enforced, at a small hardware cost (< 3%). *ASAP* also reduces memory traffic to persistent memory by 38% and 48%, compared with the state-of-the-art hardware undo and redo logging techniques, respectively. *ASAP* is robust against increasing persistent memory latency, making it suitable for both fast and slow persistent memory technologies.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; **Processors and memory architectures**; • **Hardware** → **Non-volatile memory**; **Memory and dense storage**.

KEYWORDS

Non-volatile memory, memory persistency, hardware logging

ACM Reference Format:
Ahmed Abulila, Izzat El Hajj, Myoungsoo Jung, and Nam Sung Kim. 2022. ASAP: Architecture Support for Asynchronous Persistence. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3470496.3527399

1 INTRODUCTION

Persistent memory such as Intel Optane DC Persistent DIMM has blurred the boundary between main memory and storage, providing not only the byte-addressability and latency of DRAM but also the persistency of storage devices [30, 38, 42, 51]. The integration of persistent memory in systems enables programmers to manipulate persistent data structures at a smaller granularity than what is possible with traditional block-based devices [72]. Such desirable properties with much lower bit per cost than DRAM has driven the adoption by datacenters including Microsoft Azure [9].

Programming for a system with persistent memory typically involves grouping related write operations together in an *atomic region* with atomic durability semantics. To guarantee the atomic durability of atomic regions, Write-Ahead Logging (WAL) has been commonly used [25]. WAL consists of two key persist operations: log persist operations (LPOs) and data persist operations (DPOs) [52]. LPOs flush log entries to persistent memory before making the data persistent. The log entries ensure that a consistent state can be recovered if a crash occurs before all the data written in an atomic region has persisted. On the other hand, DPOs write back the actual data modified in the atomic region to persistent memory.

To support WAL for persistent memory, software only solutions [13, 16, 27, 40, 64] as well as hardware-assisted solutions [20, 33, 36, 54, 61] have been proposed. The disadvantages of the software-only solutions are that they offload the complexity of correctly managing logs to the software and place persist operations on the critical path of execution [13, 24, 41, 57, 64]. In contrast, the hardware-assisted solutions can initiate persist operations in a manner that is transparent to the software, and they can complete these

*Ahmed Abulila is currently affiliated with Microsoft Corporation.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527399

operations in the background, overlapping them with the execution of other instructions. These advantages make hardware logging solutions attractive.

Prior hardware logging solutions commit atomic regions synchronously. That is, although persist operations are overlapped with execution of instructions *within* an atomic region, once the end of the region is reached, all outstanding persist operations required for the region to commit must complete before instruction execution may proceed past the region. For undo logging [36, 61], a region commits when all LPOs and DPOs complete. Therefore, all LPOs and DPOs are performed synchronously to ensure that the region commits synchronously. For redo logging [33], a region commits when all LPOs complete. Therefore, DPOs can be performed asynchronously, but LPOs are performed synchronously to ensure that the region commits synchronously. In both cases, waiting for synchronous persist operations (LPO or DPO) at the end of an atomic region causes atomic regions to incur high latency. Such high latency is undesirable especially for datacenter applications where tail latency of services is of increasing concern [4].

There is a key reason why hardware-assisted solutions commit atomic regions synchronously. If atomic regions are allowed to commit asynchronously, i.e., if instruction execution is allowed to proceed past the end of an atomic region before the region has committed, it may result in violations of control and data dependencies between atomic regions. Hence, it runs the risk of a later region committing before an earlier one does, or, in a multi-threaded context, a consumer region committing before the corresponding producer does. Such cases leave the data in an unrecoverable state in case of a crash.

To address such limitations, we propose **A**rchitecture **S**upport for **A**synchronous **P**ersistence (*ASAP*). *ASAP* is a hardware logging solution for persistent memory that allows atomic regions to commit *asynchronously*. To prevent violations of control and data dependencies between atomic regions, *ASAP* tracks and enforces control and data dependencies between atomic regions in hardware to ensure that the regions commit in the proper order. By allowing atomic regions to commit asynchronously, *ASAP* does not need to wait at the end of an atomic region for outstanding LPOs or DPOs to complete, which reduces the latency of atomic regions. As such, both LPOs and DPOs are performed asynchronously. *ASAP* is based on undo logging, but the principles underlying *ASAP* can also by applied to enable asynchronous commit for redo logging. In addition to reducing the latency of atomic regions, *ASAP* provides various optimizations for reducing persistent memory traffic that are particularly effective in combination with asynchronous persist operations. *ASAP* also provides a mechanism for achieving synchronous persistence if it is needed, such as if an I/O operation depends on an atomic region committing.

Our evaluation shows that *ASAP* outperforms the state-of-the-art hardware undo and redo logging techniques by 1.41× and 1.53×, respectively, while achieving 0.96× the ideal performance when no persistence is enforced. The size of hardware structures needed to support *ASAP* is less than 3% of typical CPU chip size, and *ASAP* does not require any major hardware changes, such as to the coherence protocol or the cache replacement policy. *ASAP* reduces memory traffic to persistent memory by 38% and 48%, compared with the state-of-the-art hardware undo and redo logging techniques, respectively. Although reducing persistent memory traffic does not significantly improve performance of a single application because the persist operations are asynchronous, it still benefits other metrics such as the lifetime of the persistent memory or throughput of multiple co-running memory-intensive applications. Finally, *ASAP* is robust against increasing persistent memory latency, which makes it is suitable for both fast and slow persistent memory technologies.

2 BACKGROUND

2.1 Write-Ahead Logging

Write-Ahead Logging (WAL) has been widely used to support atomic durability of updates to persistent memory. We refer to a code region containing a group of writes that need to be atomically durable as an *atomic region*. WAL maintains a log in persistent memory that stores the information needed to recover the data to a consistent state in case a crash occurs in the middle of an atomic region's execution [25]. Although WAL guarantees atomicity and durability, it does not guarantee isolation in the presence of multiple threads. Since each application has different isolation requirements, isolation is typically managed by software. The software can enforce isolation by nesting atomic regions inside critical sections guarded by locks. That is, high latency atomic regions translate into high latency critical sections and consequently more lock contention. The latency overhead of persist operations is therefore harmful for concurrency.

2.2 Software vs. Hardware Logging

Software-only solutions for WAL require software to manage logs by including persist instructions in the code such as flush/writeback instructions and memory fences. They place expensive persist operations on the critical path of execution, incurring a significant performance penalty. To address these limitations, hardware-assisted solutions for WAL [20, 33, 36, 54, 61] perform the persist operations in the background, overlapping them with the execution of subsequent instructions. Figure 1 evaluates the impact of persist operations on throughput in the software-only approach (see Section 6.5 for methodology). The results show that compared to when no persistence is enforced (NP), DPOs ("DPO Only") reduce throughput to 0.58× (geomean), and LPOs ("LPO & DPO") further reduce throughput to 0.31× (geomean). These results motivate the use of hardware-assisted solutions to overlap persist operations with subsequent instructions.

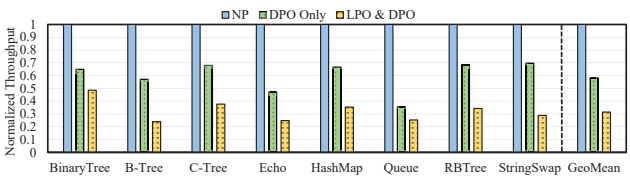


Figure 1: Overhead of LPOs and DPOs in a software approach

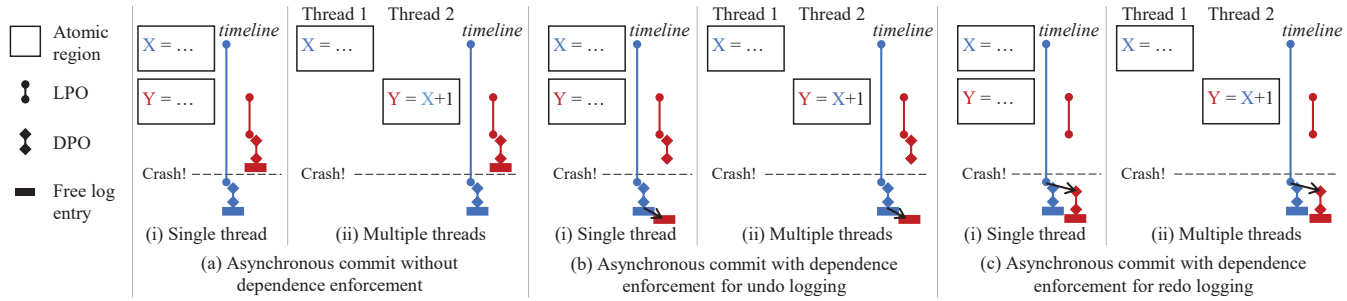


Figure 2: Examples demonstrating asynchronous commit with and without dependence enforcement

2.3 Synchronous Commit

Prior hardware logging solutions [33, 36, 54, 61] commit atomic regions synchronously. With undo logging [36, 61], when persistent data is modified, LPOs are initiated to log the old values of the modified data. Once an LPO completes, the corresponding DPO is initiated to make the new value of the data persistent in place. An atomic region is considered committed when all its DPOs complete. If a crash occurs before all the DPOs complete, then the old values are restored from the log upon recovery. Since all the DPOs must complete for the atomic region to commit, committing the region synchronously requires all LPOs and DPOs to be performed synchronously with respect to the end of the region.

In contrast, with redo logging [33], when persistent data is modified, LPOs are initiated to log the new values of the modified data, while the data itself retains the old value. Subsequent reads to the modified data are redirected to the log. Once the atomic region ends and all its LPOs have completed, the DPOs are initiated to update all the atomic region's modified data with the new values from the log. An atomic region is considered committed when all its LPOs complete. If a crash occurs before all the LPOs complete, the log is simply discarded and the old data values are retained. If a crash occurs after all the LPOs complete but before all the DPOs complete, the DPOs can be re-initiated from the log upon recovery. Since only the LPOs must complete for the atomic region to commit, committing the region synchronously requires only the LPOs to be performed synchronously with respect to the end of the region. The DPOs can be performed asynchronously.

Synchronous persist operations, whether LPOs and DPOs for undo logging or just LPOs for redo logging, force instruction execution to wait at the end of an atomic region which causes atomic regions to incur high latency. In a multi-threaded context, if the atomic regions are nested inside of critical sections, this high latency can also be harmful for concurrency. To eliminate this latency, we propose to make both LPOs and DPOs asynchronous by allowing the atomic regions to commit asynchronously.

3 ASYNCHRONOUS COMMIT

We propose a hardware-assisted logging solution that allows atomic regions to commit asynchronously in order to allow both LPOs and DPOs to be asynchronous. The challenge with committing atomic regions asynchronously is that it may result in violations of control and data dependencies between atomic regions which would leave the data in an unrecoverable state in case of a crash [2]. We address this challenge by tracking and enforcing dependencies between atomic regions in hardware. We demonstrate this challenge and how it is addressed via examples illustrated in Figure 2.

Figure 2a illustrates two examples where atomic regions are committed asynchronously (LPOs and DPOs are both asynchronous) without dependences being tracked and enforced. Figure 2a-i features a single thread, where the atomic region writing Y is control dependent on the atomic region writing X. Figure 2a-ii features multiple threads, where the atomic region writing Y is data dependent on the atomic region writing X. In both examples, the atomic region writing X initiates an LPO and does not wait for it. The LPO is performed asynchronously. Subsequently, the atomic region writing Y initiates an LPO which is performed asynchronously and completes, followed by a DPO which is also performed asynchronously and completes. A crash then occurs before the LPO of X completes. With both undo logging and redo logging, the data is left in an inconsistent state because X's new value has not persisted and is lost, whereas Y's new value *has* persisted and its old value cannot be restored. This example demonstrates how committing atomic regions asynchronously could lead to violations of control and data dependences between them.

To avoid the problem demonstrated in the previous example, we must ensure that Y's old value is not lost until X's new value has persisted. In general, if an atomic region is control or data dependent on an earlier atomic region, we must ensure that the later region's old values are not lost until the earlier region's new values have persisted. For undo logging, as illustrated in Figure 2b, we can ensure that the later region's old values are not lost by delaying freeing the later region's log entries until the earlier region's DPOs have completed. For redo logging, as illustrated in Figure 2c, we can ensure that the later region's old values are not lost by delaying the later region's DPOs until the earlier region's LPOs have completed.

In this paper, we choose to support asynchronous commit using undo logging. In prior works which rely on synchronous commit, the main advantage of redo logging over undo logging is that it supports asynchronous DPOs, whereas undo logging requires synchronous DPOs. With asynchronous commit, this advantage is no longer relevant because all persist operations are asynchronous. However, undo logging still has the advantages of performing DPOs more eagerly and not requiring reads to modified evicted data to be redirected to the log. We choose undo logging because of these advantages, however, the principles underlying our design can also by applied to enable asynchronous commit for redo logging.

While some of the techniques and insights from *ASAP* may resemble those from relaxed persistency models and from hardware transactional memory, *ASAP*'s contribution is orthogonal. We discuss how *ASAP* differs from relaxed persistency models and hardware transactional memory in Section 8.

4 ASAP DESIGN

This section describes the key design components of *ASAP*. Section 5 includes more details and discussion.

4.1 Hardware Assumptions

We target a multi-core processor sharing access to multiple memory controllers and unified LLCs. The memory organization is heterogeneous with each memory controller that can be connected to both DRAM and persistent memory modules [55]. The Write Pending Queue (WPQ) of each memory controller is considered part of the persistence domain while DRAM and caches are not. Including the WPQ in the persistence domain is consistent with modern systems where Asynchronous DRAM Refresh (ADR) [28, 59] is used to ensure that pending WPQ entries are made persistent on power failure. Accordingly, a persist operation is considered complete when it is accepted by the WPQ [66].

4.2 Software Interface

Like prior hardware-assisted solutions [20, 33, 36, 54, 61], *ASAP* provides a software interface. The interface is simple, only requiring programmers to indicate the beginning and end of atomic regions. Both LPOs and DPOs are initiated automatically, freeing the programmer from this burden.

The software interface for *ASAP* is shown in Table 1. `asap_init()` initializes *ASAP*'s metadata at thread entry. `asap_malloc()` and `asap_free()` allocate and deallocate persistent data, respectively [27, 64]. `asap_begin()` and `asap_end()` begin and end an atomic region, respectively. Nested atomic regions are permitted and are flattened by the hardware.

Table 1: *ASAP*'s software interface

API/Primitive	Description
`asap_init()`	*ASAP* initialization
`asap_malloc()`	Allocate persistent data
`asap_free()`	Deallocate persistent data
`asap_begin()`	Begin a new atomic region
`asap_end()`	End the current atomic region

ASAP's atomic regions guarantee atomic durability, but not isolation. For multi-threaded programs, programmers are required to nest conflicting atomic regions in critical sections guarded by locks. This requirement is similar to prior hardware-assisted logging approaches.

The programming burden imposed by *ASAP* is light because the functions in Table 1 are standard operations performed in any persistent memory programming interface. Moreover, wrapper libraries or simple code-generation could assist with porting legacy applications. Some prior works [36, 61] impose a heavier programming burden by requiring programmers to explicitly initiate some persist operations and enforce ordering.

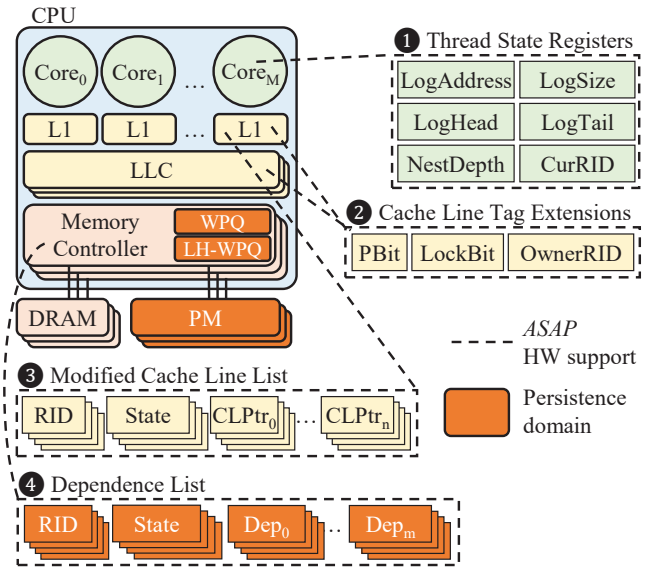


Figure 3: *ASAP* hardware extensions

4.3 Hardware Extensions

ASAP requires small hardware changes to carry out the logging and dependence tracking activities. These changes are depicted in Figure 3. *ASAP* does not require any changes to major hardware components, such as the cache replacement mechanism [33] or the coherence protocol [61]. The hardware changes required by *ASAP* are described at a high level in this subsection, and in more detail throughout the rest of the section.

Thread State Registers: These per-thread registers (❶) assist with log management and are described in Section 4.4.

Cache Line Tag Extensions (Tag Extensions): Cache lines are extended with fields (❷) that assist with executing persist operations on the cache line and detecting data dependences. These fields are described in Section 4.6.

Modified Cache Line List (CL List): This list (❸) tracks which cache lines have been modified by an atomic region. It helps ensure that all the region's persist operations complete before the region commits. The list is part of the L1 cache.

Dependence List (Dependence List): This list (❹) tracks which atomic regions are still active and the atomic regions that they depend on. It helps ensure that all an atomic region's dependencies have been resolved before its log is freed. The list is part of the memory controller and part of the persistence domain because it is needed during recovery (see Section 5.5). The *CL List* and *Dependence List* together comprise an atomic region's state. The different states that an atomic region goes through are illustrated in Figure 4, with more details described throughout the section.

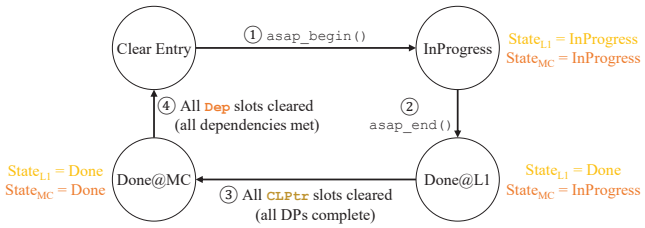


Figure 4: Atomic region state diagram

Log Header Write Pending Queue (LH-WPQ): The LH-WPQ is like the WPQ, but for the metadata of LPOs. Like the WPQ, it is part of the persistence domain. Its function is described in Section 5.5.

4.4 Initializing the Thread State Registers

ASAP uses a per thread log buffer to enhance scalability [52]. Invoking `asap_init()` at thread entry allocates a log buffer for the thread and initializes the thread state registers:

- `LogAddress`: the address of the thread's log buffer
- `LogSize`: the size of the log buffer
- `LogHead`: the index of the head of the log
- `LogTail`: the index of the tail of the log
- `CurRID`: the id of the currently active atomic region, or the latest active if no atomic region is currently active
- `NestDepth`: the nesting depth of the atomic regions (used for flattening atomic regions)

ASAP treats the allocated log buffer in memory as a circular buffer. If the log overflows, the hardware signals an exception, which is handled by allocating more log space. The programmer can also specify an initial buffer size by passing an optional parameter to `asap_init()`.

4.5 Beginning an Atomic Region

When `asap_begin()` is invoked, the hardware increments `NestDepth` and checks if the atomic region is a top-level region (`NestDepth=1`) or a nested one (`NestDepth>1`). If the atomic region is top-level, the hardware does the following:

- Increments the thread's `CurRID`
- Creates an entry for the atomic region in the *CL List*, initializing `State` to `InProgress` (①)
- Creates an entry for the atomic region in the *Dependence List*, initializing `State` to `InProgress` (①)
- If the previous atomic region (`CurRID-1`) is still in the *Dependence List*, adds it to one of the current atomic region's `Dep` slots to capture the control dependence

4.6 Handling Accesses to Persistent Memory

When memory is allocated with `asap_malloc()`, the memory allocator sets a bit in the page table to mark the allocated data as persistent. If this page table bit is set when a cache line is brought into the cache, the cache line's `PBit` is set to mark it as a persistent cache line. Accesses to persistent cache lines are treated as follows.

4.6.1 First Write (Initiating LPOs). When an atomic region writes to a persistent cache line for the first time (the cache line's `OwnerRID` is different from the thread's `CurRID`), the hardware does the following:

- Sets the cache line's `LockBit`
- Sets the cache line's `OwnerRID` to `CurRID`
- Increments the thread's `LogTail` to allocate a log entry for the cache line
- Initiates an LPO to log the old cache line value

Instruction execution then proceeds after the write while the LPO happens in the background. When the LPO completes, the cache line's `LockBit` is reset. The `LockBit` is used to ensure that no eviction or DPO takes place until the LPO completes.

4.6.2 All Writes (Initiating DPOs). On every write to a persistent cache line by an atomic region (including the first write), a pointer to that cache line is added to one of the atomic region's `CLPtr` slots in its *CL List* entry if one does not already exist. These slots track which DPOs still need to be performed. *ASAP* does not initiate a DPO for every single write to a cache line. Instead, a DPO is initiated either when four updates to other cache lines have been made or when the atomic region ends, provided the LPO initiated by the first write to the cache line has completed. Waiting for four updates to other cache lines to be made before initiating a DPO helps coalesce consecutive DPOs of the same cache line to reduce persistent memory traffic, but without waiting for too long such that all the `CLPtr` slots get occupied. The number four is empirically determined, as no benefit has been observed a distance larger than four. Once a DPO completes, the corresponding `CLPtr` slot is cleared. In the rare case that all `CLPtr` slots are occupied and a new one is needed, the hardware stalls until one becomes available, i.e., the corresponding DPO completes.

4.6.3 All Reads and Writes (Tracking Dependencies). On every read and write to a persistent cache line by an atomic region, if the cache line is owned by another atomic region (the cache line's `OwnerRID` is different from the thread's `CurRID`), the hardware adds `OwnerRID` to one of the current atomic region's `Dep` slots in the atomic region's *Dependence List* entry to capture the data dependence. The `Dep` slots are used to track whether an atomic region's dependencies have been satisfied before the atomic region commits (details in Section 4.8). If the access is a write, the current atomic region becomes the new owner of the cache line (as mentioned in Section 4.6.1). If all `Dep` slots are occupied, the hardware stalls until one becomes available (the corresponding atomic region commits). Note that since the `OwnerRID` is tracked at the cache line granularity, false sharing of cache lines may lead to spurious dependences. Alternatively, one could avoid these spurious dependences by tracking the `OwnerRID` at a finer granularity, however, this approach would require more hardware overhead.

4.7 Ending an Atomic Region

When `asap_end()` is invoked, the hardware decrements `NestDepth` and checks if the ending atomic region is a top-level atomic region (`NestDepth=0`). If the atomic region is top-level, the `State` in the atomic region's *CL List* entry is set to `Done` (②). This state means that the atomic region is not expecting any more `CLPtr` slots (no more writes). The instruction execution then proceeds past the end of the atomic region, while the remaining atomic region commit actions happen asynchronously.

4.8 Committing the Region Asynchronously

When all CLPtr slots of an atomic region are cleared (all DPOs are complete), the hardware checks if the State in the atomic region's *CL List* entry is set to Done (no more writes). If so, the hardware removes the atomic region's entry from the *CL List*, and sets the atomic region's State in its *Dependence List* entry to Done (③). This state means that all the atomic region's modified cache lines have persisted. The remaining step is to ensure that all the atomic region's dependencies have been met before freeing it undo log.

When all Dep slots of an atomic region are cleared (all dependencies met), the hardware checks if the State in its *Dependence List* entry is set to Done (all cache lines persisted). If all dependencies are met and all cache lines are made persistent, the hardware performs the following actions:

- Frees the atomic region's log entries
- Clears the region's entry in the *Dependence List* (④)
- Broadcasts to all other region entries in the *Dependence List*s in the memory controllers that the atomic region has completed to clear any corresponding Dep slots

The atomic region is thus considered to be committed.

5 IMPLEMENTATION DETAILS AND DISCUSSION

This section discusses certain implementation details of the design described in Section 4.

5.1 Optimizing Persistent Memory Traffic

ASAP applies three key optimizations to reduce persistent memory traffic: LPO dropping, DPO coalescing, and DPO dropping. The latter two are particularly effective in combination with asynchronous persist operations. These optimizations are not intended to improve latency because persist operations are asynchronous, so their latency is not on the critical path of execution. However, reducing memory traffic still benefits other metrics such as the lifetime of the persistent memory. The optimizations are described in this section and evaluated in Section 7.2.

LPO dropping: If an atomic region's LPO is still in the WPQ when the atomic region commits, there is no longer a need to send the LPO to persistent memory. Therefore, *ASAP* safely drops the LPO from the WPQ, thereby reducing traffic to persistent memory. This optimization is also applied in other works [61].

DPO coalescing: Consecutive DPOs of the same cache line in the same atomic region are coalesced into one DPO. This optimization is described in Section 4.6.2. This optimization is particularly effective in combination with asynchronous DPOs. If DPOs were synchronous, it is desirable to initiate the DPOs as soon as possible to minimize idle time, rather than wait for potential coalescing opportunities that may not arise.

DPO dropping: An atomic region's DPO may still be in the WPQ when a later region's LPO for the same cache line arrives. In this case, the DPO from the earlier region and the LPO from the later region contain the same data. Therefore, *ASAP* safely drops the DPO from the WPQ, thereby reducing persistent memory traffic. The DPO can be found using the contents of the LPO, which includes the address of the DPO. This optimization is particularly effective in combination with asynchronous DPOs. If DPOs were synchronous, there would be more time between them and the LPOs from subsequent atomic regions, so the opportunity for this optimization is less likely to arise.

5.2 Interaction with Synchronous Persistence

Since *ASAP* commits atomic regions asynchronously, it does not provide guarantees for when atomic regions commit, but only guarantees that they commit in the proper order relative to each other. In some cases, synchronous commit may be desired for an atomic region, such as to ensure that the region commits before an I/O operation that depends on it. For such situations, *ASAP* provides a special instruction `asap_fence()` that blocks until the last atomic region executed by a thread has committed, and consequently all prior regions that this region depends on. The programmer can therefore call `asap_fence()` just before the I/O operation of interest. For example, if the application needs to print a confirmation after a batch of updates has been completed, the application can call `asap_fence()` after the batch of atomic regions execute to ensure that they all commit before printing the confirmation. On the other hand, if the application needs to print a confirmation after every update, then `asap_fence()` needs to be called after every region. Note however that the I/O operation may come much after the atomic region. In this case, the commit will still be asynchronous with respect to the atomic region, but it will be synchronous with respect to `asap_fence()`.

5.3 Tracking Dependencies Across Evictions

In the rare case that a persistent cache line is evicted from the LLC while the atomic region that owns it is uncommitted (the cache line's OwnerRID is still in the *Dependence List*), the cache line's OwnerRID is saved to be reloaded when the cache line is reloaded. Saving and reloading the OwnerRID helps track data dependencies across LLC evictions. The ability to track dependencies across LLC evictions allows us not to set limits on the atomic region's memory footprint, and not to have to change the replacement policy.

To save the OwnerRID of persistent cache lines across LLC evictions, a small buffer in DRAM is used. It is fine to allocate the buffer in DRAM, not persistent memory, because the OwnerRID does not need to be persistent since it is not needed for recovery. It is only needed at execution time to track data dependencies between uncommitted atomic regions.

When a cache line is loaded from persistent memory, the memory controller concurrently checks if it has an associated OwnerRID in the DRAM buffer. If so, the memory controller checks if the OwnerRID is still in the *Dependence List*. If not, the OwnerRID is discarded. Otherwise, the OwnerRID is kept with the cache line so that future data dependencies on the atomic region can be detected.

To avoid turning every single request to persistent memory into multiple memory requests, *ASAP* uses a hardware-based non-counting bloom filter (BF) to identify if a concurrent access to the DRAM buffer is required. The filter is updated if a cache line is evicted while its OwnerRID is still active. The filter is cleared whenever the *Dependence List* becomes empty. Since there are no uncommitted regions at this point, dependencies on previously

evicted cache lines do not need to be tracked, so clearing the filter is safe.

5.4 Dependencies via Non-persistent Memory

ASAP tracks dependences between atomic regions by tracking the ID of the region that last wrote to a persistent cache line. However, if an atomic region writes to a non-persistent cache line, the region ID is not tracked. Hence, *ASAP* does not capture a data dependence between an atomic region that writes to a non-persistent memory location and another region that accesses that same location.

The reason for not tracking dependences via non-persistent memory, aside from it being prohibitively expensive, is that it is not a common case. Any non-persistent data written by an atomic region is likely to be intermediate data used within that region. On the other hand, data that is written by an atomic region with the intention of being read by another region is likely to be needed on crash and recovery, and therefore it will likely be persistent data. In the rare case that a programmer needs to write non-persistent data in one region and read it in another, the programmer can simply allocate that non-persistent data in persistent memory and free it later. In all the benchmarks we used, which are taken from prior work (see Section 6.4), we found none that needed to write non-persistent data in one region and read it in another.

5.5 Log Structure and Management

Log Structure: *ASAP* uses a distributed log where each thread maintains its own log. Using a distributed log avoids contention on updating the log in multi-threaded applications [52]. An atomic region's log space is divided into multiple records. Each record has a single metadata entry (`LogHeader`) and multiple data entries, as shown in Figure 5a. The `LogHeader` contains the `RID` and `State` of the current atomic region and the addresses of each data entry in the record. The `LogHeader` thus occupies a single cache line. This log structure is commonly used [33, 36] because it reduces the number of persistent memory writes needed to make log entries persistent. In particular, the addresses of multiple log entries are made persistent with a single cache line write.

Adding Entries to the Log: Each uncommitted atomic region keeps the `LogHeader` of its latest record in *LH-WPQ* along with the `LogHeaderAddr`, which points to the physical address of the `LogHeader` in memory (see Figure 5b). When an atomic region performs an LPO, *ASAP* sends the logged value to the WPQ and the address to the *LH-WPQ* in the corresponding field in the `LogHeader`. Once all the log entries in a record are filled, the atomic region's `LogHeader` is moved to the WPQ to be written at the corresponding `LogHeaderAddr`. A new `LogHeader` is created in the *LH-WPQ* for the atomic region's next log record.

Freeing the Log on Commit: When an atomic region commits, the region's log records are deallocated from the circular log buffer. The deallocation happens by updating the `LogHead` in the Thread State Registers to point after the atomic region's log records. The end of an atomic region's log records can be inferred from the final log record's `LogHeaderAddr` in the LH-WPQ.

Crash and Recovery: In case of a crash, the WPQ, LH-WPQ, and active entries in the *Dependence List* are flushed to persistent memory. To recover from the crash, *ASAP* uses the persistent *Dependence List* entries to infer the order in which the uncommitted atomic regions should be undone. The *Dependence List* contains the dependencies of each uncommitted atomic region. These dependencies are used to construct a directed acyclic graph of dependencies which is traversed to extract the happens-before order of the uncommitted atomic regions. *ASAP* then finds the log records of each of the atomic regions and restores the old data values.

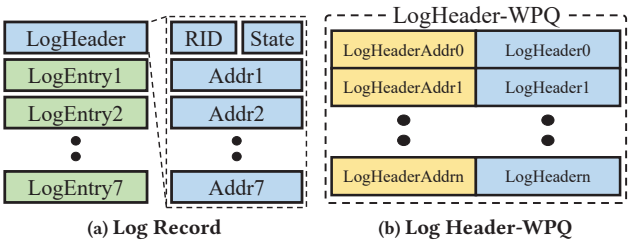


Figure 5: *ASAP* log organization and the log header-WPQ

5.6 Representing the Atomic Region ID

The atom region ID (`RID`) consists of two parts: the `ThreadID` which differentiates atomic regions from different threads, and the `LocalRID` which differentiates atomic regions from the same thread. The inclusion of the `ThreadID` in the `RID` removes the need to synchronize across threads when assigning atomic region IDs [23]. The `RID` is often used to look up the atomic region's *Dependence List* entry in the memory controller. Since there could be multiple memory controllers, we use the LSBs of the `LocalRID` to decide which memory controller to store an atomic region's *Dependence List* entry to, and to find it later on when performing a lookup.

5.7 Context Switching

On a context switch, the Thread State Registers described in Section 4.4 are saved as part of the process state. Additionally, the entry of the suspended thread in the Modified Cache Line List is cleared after completing the persist operations for each `CLPtr` slot. Clearing this entry is important because the thread may be re-scheduled on a different core. Once the thread is rescheduled, it can safely continue executing any remaining operations of its `In Progress` atomic region.

5.8 Example

Figure 6 illustrates an example of how *ASAP* handles two concurrent atomic regions running on two different cores with data dependence between them. Since both atomic regions update the location `A`, a lock `x` is used to guarantee isolation.

In Figure 6a, atomic region `R1` has already started and updates the location `A` with the value `A'` which initiates an LPO on the old value of `A`, sets the cache line's `LockBit`, and adds the cache line to `R1`'s *CL list* entry. On the other core, atomic region `R2` is initiated by calling `asap_begin()` which initializes an entry in the *CL list* of that core and an entry in the *Dependence List* in the memory controller.

In Figure 6b, the LPO for `A` has already persisted and cleared the `LockBit`, and an LPO for `B` has already been initiated. The atomic

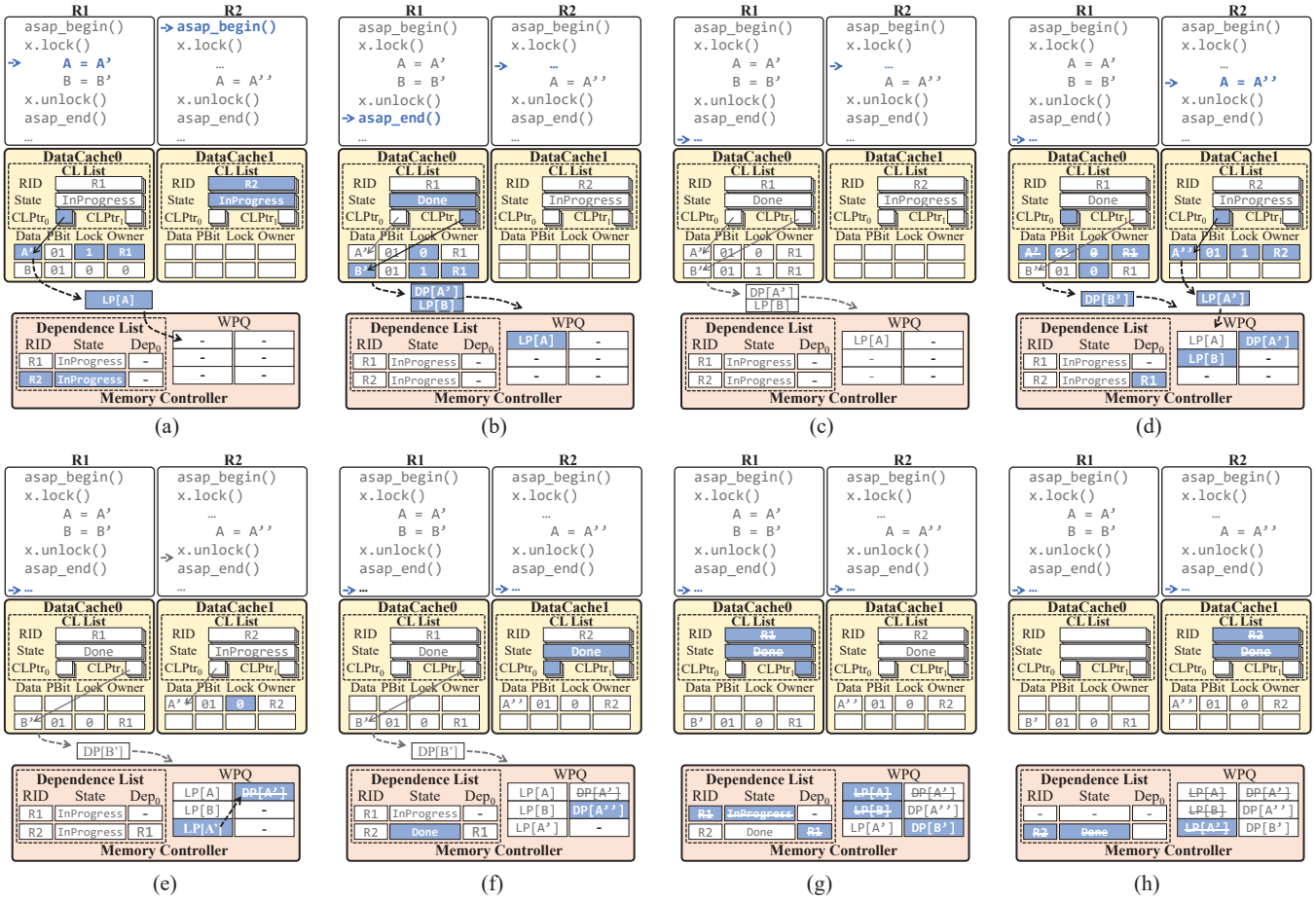


Figure 6: An example illustrating how *ASAP* handles two concurrent atomic regions with data dependence between them.

region R1 executes asap_end(), which sets the State in the *CL List* entry to Done and begins draining its CLPtrs by initiating DPOs. DPO for A' is initiated, but not for B' because the LockBit is still set.

In Figure 6c, execution continues past R1 while the LPO for B and the DPO for A' are still not persistent. This scenario demonstrates how atomic regions commit asynchronously, and LPOs and DPOs are both asynchronous.

In Figure 6d, the LPO for B arrives, which clears the LockBit, allowing the DPO for B' to be initiated. The DPO for A' also arrives. In the meantime, R2 writes to the location A now containing A' which initiates an LPO on A''. Additionally, since the cache line was previously owned by R1, R1 is added to R2's dependence list, and the cache line's owner is updated to R2.

In Figure 6e, the LPO for A' arrives, which causes the DPO for A' to be dropped according to the DPO dropping optimization (see Section 5.1).

In Figure 6f, R2 has already ended, and its cache lines drained. The arrival of the last DPO operation in R2 marks it as done in the dependence list. However, it cannot commit yet because it has a dependence on R1, which has not completed.

In Figure 6g, the DPO on B' arrives, causing R1 to be marked as done in the dependence list. Since R1 has no dependencies, it can be committed. Since its LPOs are still in the memory controller, they can be dropped according to the LPO dropping optimization (see Section 5.1). R1 also broadcasts its completion, which causes the dependence in R2's dependence list entry to be cleared.

In Figure 6h, R2 may finally commit because its dependencies are cleared. Since its LPOs are still in the memory controller, they can be dropped according to the LPO dropping optimization (see Section 5.1).

In this example, a single memory controller is used for simplicity. However, *ASAP* supports a system with multiple memory controllers as well, where each memory controller has its own *Dependence List*. When a new atomic region begins, its *Dependence List* entry is mapped to the memory controller based on the region's RID, as mentioned in Section 5.6.

Also, in this example, when R2 accesses data with its OwnerRID set to R1, R1 is added as Dep of R2. However, it may have been the case that R1 has already completed. In general, *ASAP* uses the LSB of R1 to find the memory controller that hosts R1's *Dependence List* entry and checks whether this entry actually exists. If the entry

Table 2: System Configuration

Processor	OoO, 18 cores, 5-wide issue/retire, ROB: 224, FetchQ/IssueQ/LoadQ/StoreQ: 48/64/72/56
L1	32KB/core, 8-way, 4 cycles
L2	1MB/core, 16-way, 14 cycles
L3	8MB, 16-way, 42 cycles
Memory Controller	2 MCs, 2 channels/MC, 128 WPQ entries/channel
DRAM	DDR4-2400, 16GB, 2 channels
PM	Battery-backed DRAM
ASAP	*CL List*: 4 entries/core *Dependence List*: 128 entries/channel *LH-WPQ*: 128 entries/channel Bloom filter: 1KB/channel

Table 3: Benchmarks used in our evaluation

Benchmark	Description
BN [27, 53]	Insert/update entries in a binary tree
BT [27, 53]	Insert/update entries in a b-tree
CT [27, 53]	Insert/update entries in a c-tree
EO [10, 53]	Echo a Scalable key-value store for PM
HM [27, 53]	Insert/update entries in a hash table
Q [27, 53]	Insert/update entries in a queue
RB [27, 53]	Insert/update entries in an red-black tree
SS [22, 41]	Random swaps in an array of strings
TPCC [34, 62]	New Order transaction in TPC-C

does not exist, then R1 has already committed, so it will not be added to the dependence list of R2.

6 METHODOLOGY

6.1 Simulation

ASAP has been implemented and evaluated on gem5 [11] using the system-call emulation (SE) mode with the ARM ISA. The hardware initiated LPO and DPO mechanisms are enabled in the cache controller of the L1 cache with the support of *CL List* and Tag Extension. The *Dependence List* entries and LH-WPQ have been added to the memory controller model in gem5. The detailed system configurations and parameters are summarized in Table 2. We assume a heterogeneous main memory system that pairs a persistent memory with the DRAM. The persistent memory is configured as a battery-backed DRAM by default, but we evaluate sensitivity to slower persistent memory technologies in Section 7.3. We also evaluate sensitivity to a smaller LH-WPQ size in Section 7.4.

6.2 *ASAP*'s Overhead

The *CL List* in each core has 4 entries, and its size is 49B (8 `CLPtrs`/entry, 1 B/`CLPtrs`, 2 bits/`State`, 4 B/`RID`). The *Dependence List* has 128 entries per memory channel (4 `Dep`/entry, 4B/`Dep`, 2 bits/`State`, and 4B/`RID`). The `LH-WPQ` has 70B/entry (6B `LogHeaderAddr`, 64B/`LogHeader`). In addition, *ASAP* has 6 state registers per thread. We evaluate the area overhead using McPAT [44, 67]. Compared to a baseline with no support for hardware logging, the total area overhead is about 2.5%: 0.8% core (thread state registers, L1/L2 tag extensions, CLList) and 1.7% uncore (L3 tag extensions, DependenceList, LH-WPQ, Bloom filter). Thanks to *ASAP*'s simplicity, *ASAP* does not add any structural latency to any component of the memory hierarchy.

6.3 Baselines

We compare *ASAP* to the following four baselines.

Software Persistency (SW): This baseline uses a software-only implementation of undo-logging to enforce persistency. We use distributed logging for a fair comparison. We also hand-optimized the code to coalesce different persist operations in the same atomic region that fall on the same cache line, and to overlap persist operations when possible.

Hardware Undo-logging (HWUndo): This baseline is based on the state-of-the-art hardware undo-logging implementation [61], which performs synchronous commit. This baseline only initiates LPOs automatically and transparently to the programmer. The programmer is responsible for initiating the DPOs manually [61]. Therefore, the DPOs are inserted manually for this baseline. DPOs in the same atomic region that fall on the same cache line are coalesced, as with the SW baseline.

Hardware Redo-logging (HWRedo): This baseline is based on the state-of-the-art hardware redo-logging implementation [33], which performs synchronous commit. HWRedo performs LPOs synchronously and DPOs asynchronously.

No Persistency (NP): In this baseline, data is read from and written to persistent memory, but no atomic durability is guaranteed. In other words, no LPOs or DPOs are performed. NP is intended to show the upper limit on the performance that can be achieved.

For a fair comparison, all the baselines use the same size WPQ. Additionally, the hardware logging baselines (HWUndo and HWRedo) use on-chip persistence resources of similar size to *ASAP*'s LH-WPQ to store their logging metadata.

6.4 Benchmarks

Table 3 describes the benchmarks that are used in the evaluation. These benchmarks are selected due to their nature of stressing persistent memory update performance and are adapted from or implemented similar to the benchmarks used in prior work [1, 10, 16, 22, 33, 35, 36, 41, 53, 56, 61]. All benchmarks are thread-safe with the dataset accessible to all threads. Thread-safe benchmarks allow evaluating the interaction between persistence overhead and concurrency. The benchmarks do not use `asap_fence` in between regions because the focus of our evaluation is asynchronous persistence. If `asap_fence` is used, then *ASAP* degenerates to HWUndo.

6.5 Motivational Experiment

The motivational experiment in Figure 1 is conducted using a server with four sockets, each equipped with an Intel Xeon Gold 6140 processor and 512GB of DDR4 memory. The experiment uses the same workloads as in Table 3. `clwb` and `mfence` are used to perform persist operations.

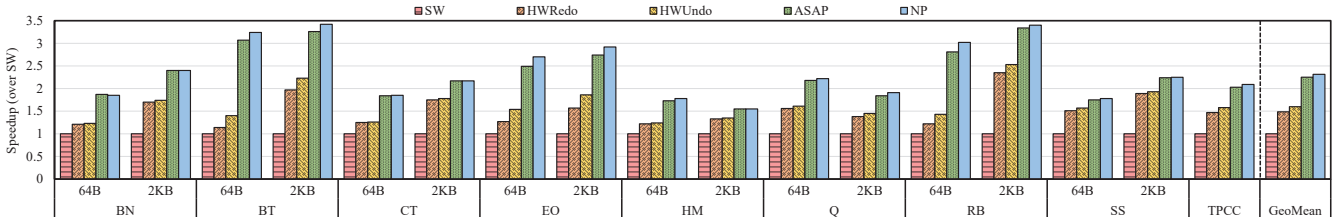


Figure 7: Performance comparison (higher is better)

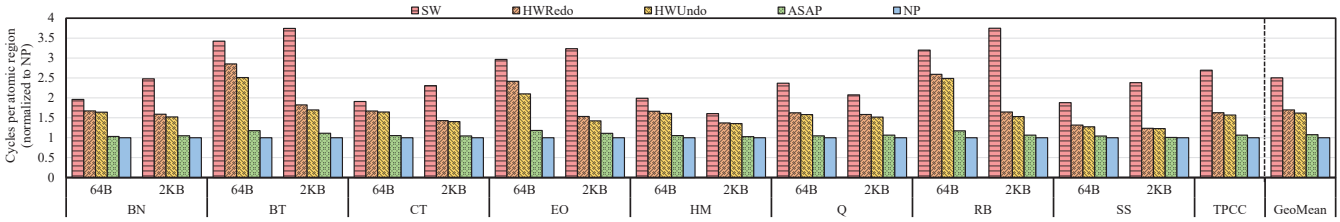
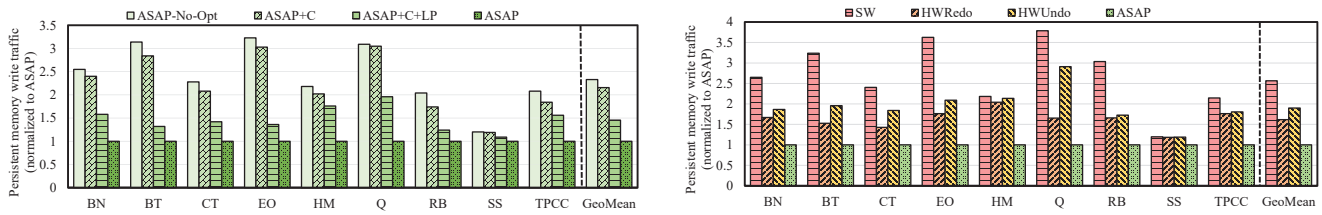


Figure 8: Normalized average number of cycles per atomic region (lower is better)



(a) Incremental improvement of *ASAP*'s memory traffic optimizations

(b) Comparison of *ASAP*'s memory traffic with other approaches

Figure 9: Persistent memory write traffic (lower is better)

7 EVALUATION

7.1 Performance

Figure 7 evaluates the speedup of HWRedo, HWUndo, and *ASAP* over SW for all benchmarks, with 64B and 2KB data sizes per atomic region. NP represents the upper bound on performance. Compared to SW, HWRedo and HWUndo improve performance by 1.49× and 1.60×, respectively. HWRedo and HWUndo are more capable than SW of overlapping LPOs with the execution of other instructions within the same atomic region. The gap between SW and HW approaches increases for larger atomic region sizes because SW has to wait on more persist operations to complete which hardware approaches can perform in the background. We note that while HWUndo outperforms HWRedo in this experiment, we show in Section 7.3 that HWRedo outperforms HWUndo for persistent memories with higher latency.

Although HWRedo and HWUndo outperform SW, there is still a considerable performance gap between them and NP, where NP is 1.56× and 1.48× faster, respectively. Since these approaches commit atomic regions synchronously, HWUndo must wait at the end of the region for LPOs and DPOs to complete, whereas HWRedo must wait for LPOs to complete. As a result, the average number of cycles per atomic region for HWRedo and HWUndo is larger than that of NP by 1.69× and 1.61×, respectively, as shown in Figure 8.

In comparison, *ASAP* achieves a speedup of 2.25× over SW, 1.52× over HWRedo, and 1.41× over HWUndo, coming very close to NP performance. NP is only 1.04× faster than *ASAP* on average. Unlike both HWUndo and HWRedo which commit atomic regions synchronously, *ASAP* commits atomic regions asynchronously, so it is capable of executing past the end of the atomic region without waiting for the LPOs and/or DPOs to complete. Therefore, the average number of cycles per atomic region of *ASAP* is only 8% higher than that of NP, as shown in Figure 8.

7.2 Memory Traffic

Recall from Section 5.1 that *ASAP* applies multiple optimizations to reduce persistent memory traffic. We evaluated the impact of these optimizations on performance and found it to be negligible because *ASAP* performs persist operations asynchronously. Nevertheless, reducing memory traffic has other benefits so this section shows traffic reduction results.

Figure 9a shows the incremental benefit of each of *ASAP*'s memory traffic optimizations. *ASAP-No-Opt* does not apply any optimizations. *ASAP+C* applies DPO coalescing, reducing traffic by ~8%. *ASAP+C+LP* additionally applies LPO dropping, further reducing

traffic by ~33%. *ASAP* additionally applies DPO dropping, further reducing traffic by ~31%.

Figure 9b compares the memory traffic of SW, HWRedo, HWUndo, and *ASAP*. HWRedo and HWUndo generate 0.63× and 0.74× the memory traffic compared with SW, respectively. HWUndo reduces the memory traffic to persistent memory by dropping LPOs from the WPQ for an atomic region that commits (see LPO dropping in Section 5.1). HWRedo takes advantage of using DRAM on commit to filter out any unnecessary DPOs to persistent memory.

In comparison, *ASAP* generates 0.62×, 0.52×, and 0.39× the memory traffic to persistent memory compared with HWRedo, HWUndo, and SW, respectively. *ASAP* further reduces the memory traffic to persistent memory via the DPO coalescing and DPO dropping optimizations, which are particularly effective in combination with asynchronous persist operations as discussed in Section 5.1.

The benchmark with the most significant memory traffic reduction compared to HWUndo is Q, as shown in Figure 9b. The Q benchmark exhibits a high amount of data dependencies across atomic regions compared to other benchmarks. Consequently, the probability of an LPO targeting the same memory location as a prior DPO is higher than other benchmarks. Hence, DPO dropping is particularly effective for this benchmark, as shown in Figure 9a.

7.3 Sensitivity to Slower Memory

Persistent memory refers to a variety of different memory technologies, ranging from fast battery-backed DRAM to other slower non-volatile memory technologies [3, 30, 38, 42, 51]. To study the impact of the latency of the persistent memory technology on our design, we vary the latency of access to persistent memory from 1× to 16× that of battery-backed DRAM. The results are shown in Figure 10.

We observe that HWRedo has lower sensitivity to the persistent memory access latency than HWUndo. The throughput of HWUndo degrades with slower memories because slow synchronous persist operations extend the critical path of atomic regions. In contrast, HWRedo asynchronously performs DPOs to the persistent memory causing it to have lower sensitivity than HWUndo to slower technologies.

In comparison, *ASAP* has a higher throughput than both HWRedo and HWUndo across different persistent memory technologies because *ASAP* does not perform any persist operations in the critical path of execution. The sensitivity of *ASAP* is closer to that of NP than HWRedo and HWUndo. Therefore, *ASAP* is robust against increasing persistent memory latency, which makes it suitable for both fast and slow persistent memory technologies.

ASAP's low sensitivity to the latency of persist operations also makes it suitable for NUMA systems where the latency of persist operations may vary. *ASAP* already supports multiple memory controllers per chip, so it can scale to multiple NUMA nodes. In a NUMA system, the *Dependence List*'s entries can be extended to include information about whether an `RID` exits as a dependence in a remote *Dependence List* or not, which makes broadcasting the completion of an atomic region more efficient.

7.4 Sensitivity to LH-WPQ Size

Recall that *ASAP* is evaluated with an LH-WPQ size of 128 entries/channel, and that HWUndo and HWRedo use structures of comparable size to store their logging metadata. We also evaluate *ASAP* with an LH-WPQ size of 16 entries/channel, and find that it performs 0.78× slower. Hence *ASAP* with 16 entries/channel still outperforms HWRedo and HWUndo with 128 entries/channel by 1.18× and 1.10×, respectively. Therefore, *ASAP* can outperform the hardware baselines that rely on synchronous persistence, while also using fewer resources for managing the logging metadata.

8 RELATED WORK

Persistency models: Several persistency models [21, 35, 39, 41, 53, 56, 60] have been defined to reason about ordering updates to persistent memory. Relaxed persistency models relax the order of updates to persistent memory at the semantic level. In contrast, our work relaxes the order of updates only at the implementation level. Semantically, writes to persistent memory in *ASAP* happen atomically within atomic regions and in-order across regions. These semantics are enforced by the logging and dependence tracking mechanisms. Hypothetically, *ASAP* could be designed to enforce a more relaxed persistency model between different atomic regions, which would change the way dependecies are tracked and enforced. Hence, persistency models and atomic durability are orthogonal concepts. Our work targets the latter.

Software Support for Atomic Durability: Managing persistent memory using a file system is an intuitive approach to exploit persistent memory features while assuring crash-consistency [17, 18, 37, 48, 63, 68, 69, 73]. To reduce performance overhead, others works provide libraries to enable fast user-mode access to in-memory data sets while guaranteeing crash-consistency [13, 14, 16, 24, 26, 31, 45, 50, 58, 64]. Section 2.2 motivates the use of hardware-assisted solutions to improve on software-only solutions. With a well-defined interface software approach, the programmer effort to adapt an application to utilize the persistent memory benefits is expected to be reasonable. On the other hand, these techniques suffer from unnecessary execution stalls because of the expected minimal hardware support [41]. In addition, the number of executed instructions dramatically increases to guarantee crash-consistency in such systems [49].

On-Chip Data Versioning: Several works use on-chip resources to contain partial updates on-chip until an atomic region completes [1, 6, 7, 23, 31, 43, 74], after which the updates are synchronously committed to persistent memory before proceeding past the end of the atomic region. For example, Kiln [74] has a non-volatile last-level cache to preserve partially updated data. Lai *et al.* [43] adds a separate non-volatile on-chip structure to avoid complex modifications to the cache hierarchy. LAD [23] avoids adding a new on-chip structure by exploiting the WPQ in the memory controller which is considered part of the persistence domain. BBB [5] has proposed a battery-backed buffers in the CPU to bring the point of persistency closer to the CPU. Intel eADR [29] can make caches part of the persistence domain, which overcomes the latency of persist operations. However, it still requires a WAL technique to provide failure-atomicity. While eADR can simplify hardware-based WAL, eADR also requires a large battery, consuming high

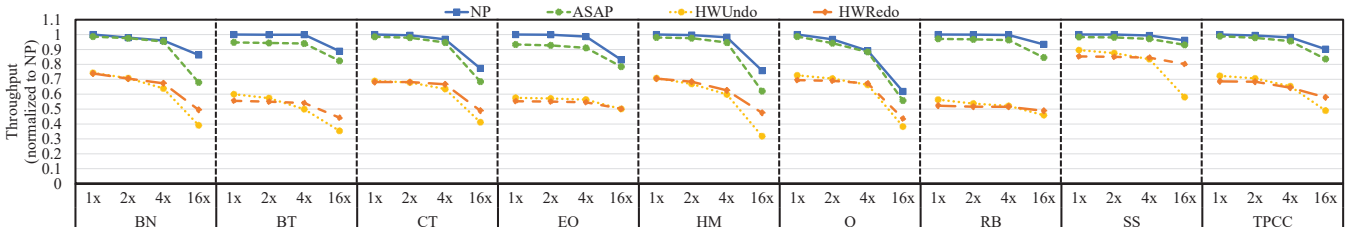


Figure 10: Sensitivity of throughput (higher is better) to memory latency

power [5, 19]. In contrast, *ASAP* can overcome the latency of persist operations and achieve near-non-persistence performance without this requirement.

Hardware Transactional Memory (HTM): Although the tracking of modified data done by *ASAP* may seem similar to HTM, they are different. HTM tracks modified data to detect conflicts between concurrent transactions. *ASAP* tracks modified data to detect dependencies between atomic regions that may run at different points in time. HTM is designed to enforce isolation, which is orthogonal to what *ASAP* does. In our work, we use locks to enforce isolation. However, one could imagine a hypothetical approach where *ASAP* regions are isolated with HTM transactions. In this case, a transaction may commit (guaranteeing isolation), but the region it isolates may commit later asynchronously (guaranteeing atomic durability). Hence, two forms of tracking would be needed. One would need to track modified data before the transaction commits to detect conflicts with concurrent transactions. One would also need to keep tracking modified data after the transaction commits to detect dependencies by later regions. Various works extend HTM with enhancements to enable crash-consistency [1, 6, 7, 32]. These works commit a transaction's atomic updates synchronously with respect to the end of the transaction. Our work commits atomic updates asynchronously with respect to the end of a critical section.

Hardware-Assisted Data Versioning in Persistent Memory: Various works have proposed hardware-assisted data versioning that versions data in persistent memory instead of on chip [12, 20, 33, 36, 49, 54, 61, 65]. These works can be classified into approaches that use redo logging [33], undo logging [36, 61], or the combination of the two [54, 65]. We discuss the difference between our work and these approaches in Section 2.3. The key distinction is that our work commits atomic regions asynchronously which enables *both* LPOs and DPOs to be performed asynchronously with respect to the end of the atomic region. LOC [49] performs asynchronous commit similar to *ASAP*, but focuses on single-threaded applications and uses redo logging. *ASAP* performs asynchronous commit for multi-threaded applications, which requires tracking data dependencies across atomic regions in different threads. *ASAP* uses undo logging, which enables more eager DPOs.

Secure Persistent Memory: The durability of data in persistent memory allows data to survive system reboots or failures, which makes persistent memory at risk of malicious attacks [15, 77]. Data encryption can provide data confidentiality and protect against probing the data out of persistent memory. Several hardware encryption mechanisms have been proposed to enable efficient and secure persistent memory systems [8, 15, 46, 47, 70, 71, 75, 76, 76, 77]. In addition, Liu *et al.* [46] have introduced a hardware mechanism to support *selective-counter atomicity* that optimizes the encryption memory traffic to persistent memory. These proposals do not alter the processor design nor the cache hierarchy and reside in the memory controller. Securing persistent memory is an orthogonal topic to this paper. *ASAP* can be efficiently applied on top of a hardware encryption mechanism to enable a secure persistent memory system. Moreover, *ASAP* can employ a similar technique as Janus [47] to overlap the persist operations with encryption. For example, *ASAP* can be integrated with DeWrite [76] while following an approach similar to the *selective-counter atomicity* [46] technique to enable an efficient persistent memory system.

9 CONCLUSION

This paper presents *ASAP*, a hardware logging scheme that allows atomic regions to commit asynchronously. Committing atomic regions asynchronously removes the need to wait for log persist and/or data persist operations at the end of atomic regions, which reduces the latency of these regions. To ensure that the atomic regions commit in the proper order, *ASAP* tracks and enforces control and data dependencies between atomic regions in hardware. Our evaluation shows that *ASAP* outperforms state-of-the-art hardware undo and redo logging techniques, which commit atomic regions synchronously. It also reduces persistent memory traffic, and is suitable for both fast and slow persistent memory technologies.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation (CNS-1705047) and National Research Foundation of Korea (2021R1A2C4001773). Nam Sung Kim has a financial interest in Samsung Electronics and NeuroRealityVision Corporation.

REFERENCES

[1] A. Joshi and V. Nagarajan and M. Cintra and S. Viglas, "DHTM: Durable Hardware Transactional Memory," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 452–465.

[2] A. Abulila, "Efficient design and optimized crash-consistency support for hybrid memory systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Champaign, IL, 2020, http://hdl.handle.net/2142/108559.

[3] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "FlatFlash: Exploiting the Byte-Accessibility of SSDs Within a Unified Memory-Storage Hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 971–985. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304061

[4] S. Akram, "Exploiting intel optane persistent memory for full text search," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 80–93. [Online]. Available: https://doi.org/10.1145/3459898.3463906

[5] M. Alshboul, P. Ramrakhyani, W. Wang, J. Tuck, and Y. Solihin, "Bbb: Simplifying persistent programming using battery-backed buffers," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 111–124.

[6] H. Avni and T. Brown, "PHyTM: Persistent Hybrid Transactional Memory," *Proc. VLDB Endow.*, vol. 10, no. 4, pp. 409–420, Nov. 2016. [Online]. Available: https://doi.org/10.14778/3025111.3025122

[7] H. Avni, E. Levy, and A. Mendelson, "Hardware Transactions in Nonvolatile Memory," in *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363*, ser. DISC 2015. Berlin, Heidelberg: Springer-Verlag, 2015, pp. 617–630. [Online]. Available: https://doi.org/10.1007/978-3-662-48653-5_41

[8] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-NVM: Persistency for Integrity-protected and Encrypted Non-volatile Memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 104–115. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322250

[9] "Next Generation SAP HANA Large Instances with Intel® Optane™ drive lower TCO," https://azure.microsoft.com/en-us/blog/next-generation-sap-hana-large-instances-with-intel-optane-drive-lower-tco/.

[10] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, "Exploring Storage Class Memory with Key Value Stores," in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, ser. INFLOW '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2527792.2527799

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[12] M. Cai, C. Coats, and J. Huang, "HOOP: Efficient hardware-assisted out-of-place update for non-volatile memory," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 584–596.

[13] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging Locks for Non-volatile Memory Consistency," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 433–452. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660224

[14] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures," *Proc. VLDB Endow.*, vol. 8, no. 5, pp. 497–508, Jan. 2015. [Online]. Available: http://dx.doi.org/10.14778/2735479.2735483

[15] S. Chhabra and Y. Solihin, "i-NVMM: A secure non-volatile main memory system with incremental encryption," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 177–188.

[16] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "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*, ser. ASPLOS XVI, Newport Beach, CA, 2011, pp. 105–118.

[17] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O Through Byte-addressable, Persistent Memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09, Big Sky, MT, 2009, pp. 133–146.

[18] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System Software for Persistent Memory," in *Proceedings of the 9th European Conference on Computer Systems*, ser. EuroSys '14, Amsterdam, The Netherlands, 2014, pp. 15:1–15:15.

[19] "From FLOPS to IOPS: The New Bottlenecks of Scientific Computing," https://www.sigarch.org/from-flops-to-iops-the-new-bottlenecks-of-scientific-computing/, Jan. 2020.

[20] E. Giles, K. Doshi, and P. Varman, "Bridging the Programming Gap Between Persistent and Volatile Memory Using WrAP," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13. New York, NY, USA: ACM, 2013, pp. 30:1–30:10. [Online]. Available: http://doi.acm.org/10.1145/2482767.2482806

[21] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Relaxed persist ordering using strand persistency," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 652–665.

[22] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 46–61. [Online]. Available: https://doi.org/10.1145/3192366.3192367

[23] S. Gupta, A. Daglis, and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 466–478. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358321

[24] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "NVthreads: Practical Persistence for Multi-threaded Applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 468–482. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064204

[25] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda, "Log-Structured Non-Volatile Main Memory," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 703–717. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hu

[26] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware logging in transaction systems," in *Proceedings of the 41th International Conference on Very Large Data Bases (VLDB'15)*, 2015.

[27] Intel, "Persistent Memory Development Kit," http://pmem.io/pmdk/, 2017.

[28] Intel Corporation, *Intel® Architecture Instruction Set Extensions Programming Reference*, Sep. 2016, 319433-025.

[29] "Micron and Intel Announce Update to 3D XPoint Joint Development Program," https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, Jan. 2021.

[30] "Micron and Intel Announce Update to 3D XPoint Joint Development Program," https://newsroom.intel.com/news-releases/micron-intel-announce-update-3d-xpoint-joint-development-program/, Jul. 2018.

[31] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-Atomic Persistent Memory Updates via JUSTDO Logging," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 427–442. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872410

[32] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon, "Unbounded hardware transactional memory for a hybrid dram/nvm memory system," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 525–538.

[33] J. Jeong, C. H. Park, J. Huh, and S. R. Maeng, "Efficient Hardware-Assisted Logging with Asynchronous and Direct-Update for Persistent Memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 520–532.

[34] E. Jones, "In-memory TPC-C Implementation," https://github.com/evanj/tpccbench, Apr. 2011.

[35] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient Persist Barriers for Multicores," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 660–671.

[36] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 361–372.

[37] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, and V. Chidambaram, "SplitFS: A File System that Minimizes Software Overhead in File Systems for Persistent Memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

[38] R. Kateja, A. Badam, S. Govindan, B. Sharma, and G. Ganger, "Viyojit: Decoupling Battery and DRAM Capacities for Battery-Backed DRAM," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 613–626. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080236

[39] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level Persistency," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 481–493. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080229

[40] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-Performance Transactions for Persistent Memories," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16, Atlanta, GA, 2016, pp. 399–411.

[41] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 58:1–58:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195709

[42] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 256–267.

[43] C.-H. Lai, J. Zhao, and C.-L. Yang, "Leave the Cache Hierarchy Operation As It Is: A New Persistent Memory Accelerating Approach," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: ACM, 2017, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3062272

[44] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual*

IEEE/ACM International Symposium on Microarchitecture, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 469–480. [Online]. Available: https://doi.org/10.1145/1669112.1669172

[45] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "DudeTM: Building Durable Transactions with Decoupling for Persistent Memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 329–343. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037714

[46] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash Consistency in Encrypted Non-volatile Main Memory Systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 310–323.

[47] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing Memory and Storage Support for Non-volatile Memory Systems," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 143–156. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322206

[48] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an RDMA-enabled Distributed Persistent Memory File System," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 773–785. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[49] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-Ordering Consistency for persistent memory," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Oct 2014, pp. 216–223.

[50] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 499–512. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064215

[51] "Micron NVDIMMs: Persistent Memory Performance," https://www.micron.com/-/media/client/global/documents/products/product-flyer/nvdimm_flyer.pdf, 2016.

[52] 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 Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

[53] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An Analysis of Persistent Memory Use with WHISPER," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 135–148. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037730

[54] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 336–349.

[55] "Intel Optane Persistent Memory Product Brief," https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html, 2019.

[56] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Minneapolis, MN, 2014, pp. 265–276.

[57] J. Ren, Q. Hu, S. Khan, and T. Moscibroda, "Programming for Non-Volatile Main Memory Is Hard," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17. New York, NY, USA: ACM, 2017, pp. 13:1–13:8. [Online]. Available: http://doi.acm.org/10.1145/3124680.3124729

[58] A. Rudoff, "Persistent memory programming," https://www.usenix.org/system/files/login/articles/login_summer17_07_rudoff.pdf, pp. 34–40, Sep. 2017.

[59] A. M. Rudoff, "Deprecating the PCOMMIT Instruction," https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction, Sep. 2016.

[60] S. M. Shahri, S. Armin Vakil Ghahani, and A. Kolli, "(almost) fence-less persist ordering," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 539–554.

[61] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 178–190. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124539

[62] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The End of an Architectural Era: (It's Time for a Complete Rewrite)," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 1150–1160. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325981

[63] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible File-system Interfaces to Storage-class Memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14, Amsterdam, The Netherlands, 2014, pp. 14:1–14:14. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592810

[64] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, CA, 2011, pp. 91–104.

[65] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye, "Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 610–623.

[66] D. Williams, "Replace pcommit with ADR or directed flushing," https://lwn.net/Articles/694134/, Jul. 2016.

[67] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 577–589.

[68] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories." in *FAST*, 2016, pp. 323–338.

[69] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 478–496. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132761

[70] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 403–415.

[71] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-Efficient Encryption for Non-Volatile Memories," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 33–44. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694387

[72] M. Zhang, K. T. Lam, X. Yao, and C.-L. Wang, "SIMPO: A Scalable In-Memory Persistent Object Framework Using NVRAM for Reliable Big Data Computing," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, pp. 7:1–7:28, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3167972

[73] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A Reliable and Highly-Available Non-Volatile Memory System," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 3–18. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694370

[74] 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," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46, Davis, CA, 2013, pp. 421–432.

[75] K. A. Zubair and A. Awad, "Anubis: Ultra-low Overhead and Recovery Time for Secure Non-volatile Memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322252

[76] 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," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 442–454.

[77] P. Zuo and Y. Hua, "SecPM: a Secure and Persistent Memory System for Non-volatile Memory," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. Boston, MA: USENIX Association, Jul. 2018. [Online]. Available: https://www.usenix.org/conference/hotstorage18/presentation/zuo