ASAP: A Speculative Approach to Persistence

Sujay Yadalam, Nisarg Shah, Xiangyao Yu, Michael Swift

University of Wisconsin-Madison
Madison, USA
Email: {sujayyadalam, nisargs, yxy, swift}@cs.wisc.edu

***Abstract*—Persistent memory enables a new class of applications that have persistent in-memory data structures. Recoverability of these applications imposes constraints on the ordering of writes to persistent memory. But, the cache hierarchy and memory controllers in modern systems may reorder writes to persistent memory. Therefore, programmers have to use expensive flush and fence instructions that stall the processor to enforce such ordering. While prior efforts circumvent stalling on long latency flush instructions, these designs under-perform in large-scale systems with many cores and multiple memory controllers.**

We propose ASAP, an architectural model in which the hardware takes an optimistic approach by persisting data eagerly, thereby avoiding any ordering stalls and utilizing the total system bandwidth efficiently. ASAP avoids stalling by allowing writes to be persisted out-of-order, speculating that all writes will eventually be persisted. For correctness, ASAP saves recovery information in the memory controllers which is used to *undo* the effects of speculative writes to memory in the event of a crash.

Over a large number of representative workloads, ASAP improves performance over current Intel systems by $2.3\times$ on average and performs within 3.9% of an ideal system.

***Keywords*-persistent memory; NVM; recoverable applications; ordering; sfence; flush; eADR;**

I. Introduction

Emerging non-volatile memory (NVM) technologies such as Intel's Optane Persistent Memory (PM) [1] pair the high performance and byte-addressability of DRAM with the durability of disks and Flash memory. These technologies enable *recoverable applications*, which store critical data structures in persistent memory to survive system crashes and power failures [2]–[4]. However, current persistent memory systems do not save the contents of the processor cache on a failure: only data in the *persistence domain* survives. On systems supporting Optane, this domain includes both data in NVM and data that has reached a memory controller [5]. As a result, developers must ensure that data has been flushed from the processor or caches to the memory controller using either non-temporal stores that write directly to memory or cache write-back instructions. These instructions incur long stalls in today's NVM systems [6].

Ensuring that applications can correctly recover from a failure is further complicated by the need to maintain data consistency. To achieve high performance, current processors reorder data written out to memory. Yet, ensuring consistency requires ordering certain updates, such as a pointer update only after the data it references. Current hardware provides a crude mechanism to enforcing order through expensive *fences* that stall processor execution until all prior flushes have reached the persistence domain.

Multi-threaded applications require persist ordering across threads, which is hard to achieve efficiently. While early studies of persistent memory usage found cross-thread dependencies rare [6], they are frequent in a new class of concurrent applications: recently developed high-performance, scalable concurrent data structures [4], [7]–[12] specifically for persistence. For these structures, efficient support of concurrent accesses and subsequent cross-thread dependencies is critical for performance.

Several prior proposals have looked to reduce the high cost of ordering by either entrusting the cache hierarchy to enforce ordering lazily [13], [14], or using buffering [6], [15]–[19] or through speculation [20]. Some of these designs do not support ordering across multiple memory-controllers [15], [20], and others [6], [14], [15], [17] suffer long *flushing stalls* while persisting writes ordered across memory controllers. Also, these designs employ conservative techniques that stall flushing to enforce ordering in the presence of cross-thread dependencies. Other designs [16] either increase the hardware complexity or require batteries for flushing data on a failure [19].

This paper introduces ***ASAP*** (A Speculative Approach to Persistence), a novel persistence architecture that provides the ability to order writes to NVM with almost no stalls even in the presence of cross-thread dependencies and on systems with multiple memory controllers. The key insight of our work is that in the absence of failure, it is fine to write data out to NVM in any order. Only when a failure occurs must the system ensure that ordering was enforced. ASAP therefore flushes writes optimistically and stores enough information to correct the contents of memory when a failure occurs. For example, when a write arrives early, ASAP allows a speculative write to update NVM, but also saves the old value at the address. If a failure occurs, ASAP can restore the old value. In effect, ASAP maintains an undo log for anything updated out of order.

While writing an undo record to NVM would be expensive

2378-203X/22/$31.00 ©2022 IEEE
DOI 10.1109/HPCA53966.2022.00070

and lead to high write amplification, ASAP avoids additional writes by leveraging Intel's Asynchronous DRAM Refresh (ADR) technology to save recovery information in the memory controller. ADR allows a small amount of data to be written back from the memory controller to NVM following a failure but before power to the processor is completely lost. On failure, ASAP uses the undo information at the memory controller, to restore NVM to the correct state.

Storing recovery information at the memory controller differs from logging employed in hardware approaches that provide atomic durability [21]–[28]. First, these hardware-assisted logging mechanisms generate additional writes to NVM as they create a log entry on every write while ASAP stores recovery information only when memory is updated speculatively. Second, these designs can incur long latency to enforce ordering between log and data writes, while ASAP can reorder writes. Third, they may require large buffers to hold out-of-place updates while ASAP performs well with small buffers. ASAP does not aim to provide atomicity and instead enforces ordering of writes efficiently. Many PM applications [4], [7]–[10] do not use transactions but still require ordering primitives from the hardware. These designs would therefore benefit from ASAP but not from the hardware transactional models. If applications do require atomicity, ASAP can be coupled with any techniques such as shadow paging or software transactions. This flexibility allows ASAP to re-order writes to NVM, speculatively update memory and use small buffers.

ASAP extends the processor core with persist buffers that queue data waiting to be flushed to NVM. Queued writes are flushed speculatively and out-of-order. Ordering dependencies across threads are tracked by augmenting the coherence protocol and are resolved through inter-core communication. A small buffer called the recovery table at the memory controller stores the recovery information.

To study the performance benefits of ASAP, we compare it against one of the prior works that supports ordering across multiple memory controllers, HOPS [6], and against BBB [19] which has close to ideal performance. We evaluate ASAP on a wide range of benchmarks including transactional applications using Intel's PMDK [29], applications written to natively use PM, data structures using the Atlas framework [30], and hand-written persistent data structures. On average, ASAP is 22.8% faster than HOPS and within 3.9% of BBB [19]. In addition, we show that ASAP's performance scales with increasing threads, and offers greater performance benefit with increasing NVM write bandwidth.

II. Background

A. Persistency models

To ensure safety and correct recovery of persistent applications, the program state stored in persistent memory must be in a consistent state. Persistency models [31], like memory consistency models, define what constitutes a consistent state and hence how a system is allowed to reorder operations. We refer to stores in the volatile memory order as *writes* and stores in the persistent order as *persists*.

ISA-level persistency models define what persist orderings a processor exposes to programs. *Strict persistency* couples the order of persists to the order of writes in volatile memory order. It avoids the necessity for barriers to express ordering but frequent ordering operations and the inability to coalesce can hurt performance. *Epoch persistency* is more relaxed as it allows some persists to be combined and/or reordered. Every thread's execution is divided into sequences of persists called epochs, which are separated by fences (also called persist barriers). Persists within an epoch can be reordered or occur in parallel, allowing persists to coalesce. *Epoch persistency* enforces two ordering constraints: (1) any two memory accesses belonging to different epochs, i.e., separated by a persist barrier, are ordered. (2) Conflicting memory accesses (accesses to the same address) within and across threads assume the persist order from their volatile memory order. The latter constraint is also called *strong persist atomicity*. *Strand persistency* is the most relaxed model. It divides a thread's execution into strands. Writes in different strands can be flushed concurrently but inter-strand dependencies can still arise due to *strong persist atomicity*.

Language-level persistency. Reasoning about persist ordering across threads can be challenging. *Acquire-Release Persistency* (ARP) is a language-level persistency model that uses explicit acquire and release synchronization primitives to specify ordering across threads [32]. When an acquire synchronizes with a release, all writes preceding the release should become durable before writes following the acquire. Ordering within a thread is achieved by using barriers.

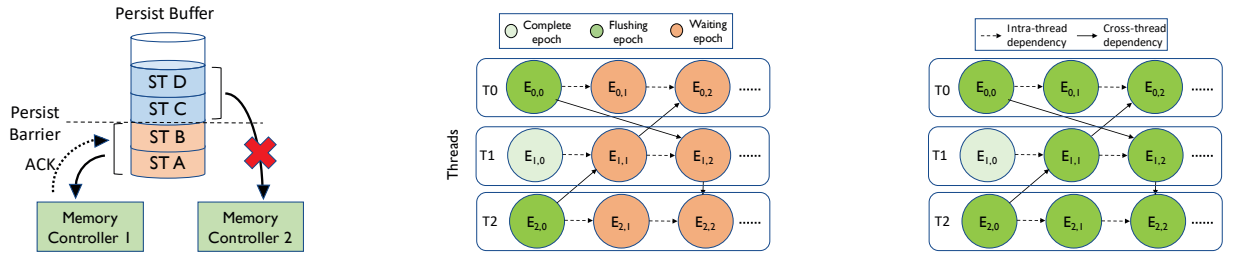
Along with the ordering that ARP enforces, *Release Persistency* (RP) imposes ordering between writes and the synchronization instructions themselves (acquire/release) [18], allowing persistent synchronization variables. Writes preceding a release are persisted before the release operation.

B. Synchronous ordering

Current Intel systems support PM through instructions for durability and ordering. Programs flush data to NVM using `clflushopt` and `clwb` to force write-back of cache lines into memory [33]. These instructions are weakly ordered, so a program that wants to enforce ordering must use a `sfence` between epochs to wait for the preceding flushes to complete, which stalls the CPU until the memory controller acknowledges the flushes. Frequent ordering hurts performance by stalling on every fence instruction [34].

C. Asynchronous DRAM Refresh (ADR) and enhanced ADR

Intel platforms that support Optane Persistent Memory provide a feature called Asynchronous DRAM Refresh [5], which flushes data queued at the memory controller to



(a) Conservative flushing in a multi memory-controller system

(b) Conservative flushing with cross-thread dependencies

(c) Eager flushing in ASAP enables dependent threads to flush concurrently

Figure 1: Conservative flushing in earlier designs versus eager flushing in ASAP

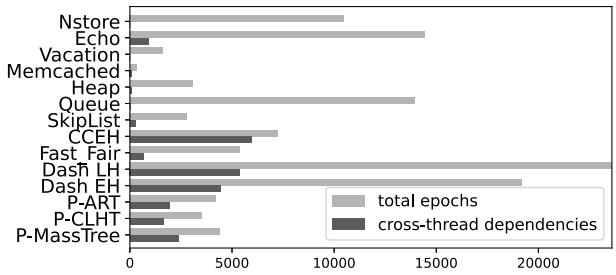


Figure 2: Number of epochs and cross-thread dependencies across 4 threads within 1 ms of execution

NVM on a power loss. Therefore, writes to NVM can be considered durable once they reach the memory controller.

Recently, Intel announced enhanced ADR (eADR) that flushes processor caches on power failure [5]. If eADR is enabled, applications do not have to issue flush instructions as all the data in the caches would be flushed to NVM in the event of a crash. The operating system must be able to handle a power fail interrupt and detect if the reserve power is insufficient to flush caches, indicating that eADR may not always guarantee data is flushed [35] [36]. However, the details of eADR are not available yet, so the implications on performance are not clear. eADR is expected to be costly as it requires a large battery to back the entire cache hierarchy [36], [37]. Adding a battery to every system supporting NVM is not generally practical or cost-effective. ASAP is able to achieve performance close to that of eADR (Section VI) without requiring additional battery.

III. Motivation

The motivation for ASAP is two-fold: (1) Avoid flushing stalls in a multi-core multi-memory controller system and (2) handle cross-dependencies efficiently.

Multi memory-controller systems. Having multiple memory controllers in a single processor is common for server-class processors, such as Intel Xeon processors (the only ones to currently support NVM). With multiple memory controllers, enforcing ordering requires ensuring that persists are ordered correctly across multiple controllers. So persists at one controller may have to wait for persists to arrive and complete at another controller. Compounding the problem, the lower bandwidth of Intel Optane memory encourages interleaving physical addresses across controllers to increase memory parallelism. Recent studies [38], [39] have shown that interleaving can improve write bandwidth by up to 5.6×. This makes it more likely that data structures will span memory controllers.

Currently available hardware and previously proposed designs either do not address this problem or are not efficient in doing so. Intel hardware tackles the problem through synchronous fence operations: the processor stalls at every epoch boundary to wait for preceding persists to complete, which increases latency and fails to utilize the available bandwidth to NVM. Many prior works such as DPO [15] and PMEM-Spec [20] do not support multiple memory controllers. Most other proposals such as LB++ [14], HOPS [6], StrandWeaver [17], and LRP [18] implement inefficient *conservative flushing* strategies. These designs order writes across memory controllers by waiting for writes in the current epoch to be acknowledged before flushing writes from later epochs. Figure 1a illustrates this case, where the second epoch writing to MC 2 must stall until the first epoch is acknowledged by MC 1. These designs incur flushing stalls when one memory controller is slower to respond than the other and leads to poor bandwidth utilization.

To address this issue, Vorpal [16] proposed distributed algorithms using vector clocks. Implementing such complex protocols can involve significant overheads: 1) high tag cost: every store needs to be tagged with a vector timestamp (a vector containing as many timestamps as cores) which can be quite large in a server with many cores, 2) communication overheads: memory controllers need to broadcast their clocks frequently as the broadcast frequency determines the rate of forward progress. There is a need for an efficient implementation for ordering persists in a multi-memory controller system.

Handling cross-thread dependencies. Efficiently ordering persists correctly across threads is important for performance. The WHISPER workload analysis [6] showed that applications had few cross-dependencies, where one thread

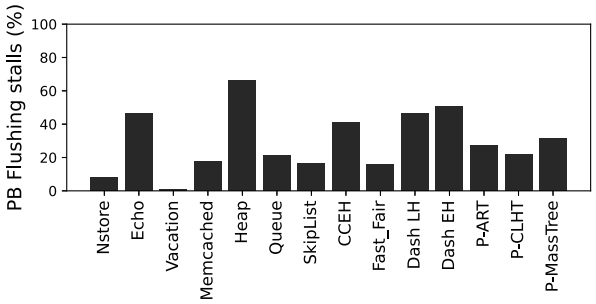
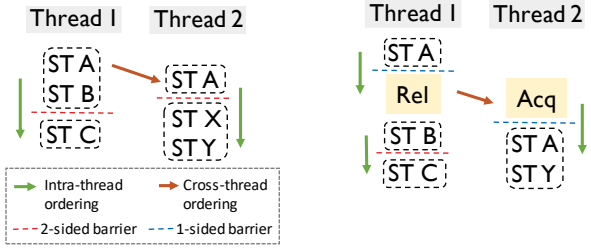


Figure 3: Percentage of persist buffer stall cycles



(a) Epoch persistency (b) Release persistency

Figure 4: Persistency model semantics provided by ASAP

persists data recently persisted by another thread. Since that work, there have been numerous efforts at producing high-performance concurrency-aware persistent data structures specifically designed for efficient multi-threaded workloads (e.g., CCEH [7]). Figure 2 shows the total number of epochs and number of cross-dependencies in 1ms of simulated program execution with release persistency. Cross-dependencies are not common in workloads in the WHISPER benchmark and PMDK-based applications such as Vacation and Memcached. However, they are frequent in new concurrent data structures such as CCEH [7], Dash [8] and RECIPE [4]. Thus, there is a necessity for a system that handles these dependencies efficiently.

To handle these dependencies, designs that employ *conservative flushing* stall flushing in the dependent thread until the earlier thread's persists complete. Figure 1b illustrates this problem. Writes in epoch $E_{1,1}$ of thread T1 cannot be flushed until source thread T2 completes flushing its epoch $E_{2,0}$. PMEM-Spec [20] speculates that persists will occur in order, but has an expensive recovery mechanism if this speculation is wrong.

We measure the frequency of such flushing stalls for various workloads with HOPS (see Section VI for workload details) to demonstrate the extent of the problem. Figure 3 shows the percent of cycles the persist buffers are blocked without flushing writes for different workloads. Overall, the persist buffers are unable to flush for 26% of cycles on average. Many of these stalls come from cross-dependencies which is why the stall cycles are higher in the new concurrent persistent data structures.

Although earlier works alleviate processor stalls by decoupling persistence from core execution with caches/buffers, they suffer from frequent flushing stalls. Such frequent stalls cause the finite buffers to fill up and exert back-pressure on the core pipeline, ultimately stalling the processor.

IV. ASAP Overview

The main design goals of ASAP are:

- *Reducing core stalls by avoiding flushing stalls*: ASAP overcomes flushing stalls by flushing writes eagerly and possibly out-of-order. This in turn reduces the number of core stalls, increases bandwidth utilization, and improves the overall performance.
- *Maximizing common case performance*: Persist ordering has to be preserved to ensure consistent state during crashes. In the absence of failures, ordering is not a necessity. Crashes are rare compared to the frequency of ordering events during program execution. ASAP, therefore, tries to improve the performance during normal operation by speculatively updating memory.
- *Handling cross-thread dependencies efficiently*: ASAP employs cross-thread dependency tracking and resolution strategies to overcome limitations of earlier works.
- *Scaling to large scale servers*: ASAP avoids global shared state, global broadcast operations, or centralized structures with per-core state in order to scale to many cores and multiple memory controllers.

ASAP is a buffered persistency system that uses a separate data path for persists. Writes are queued in a hardware buffer and are later flushed from the buffer to the memory controller. Buffering decouples the program execution from data persistence, allowing the core to continue executing instructions while the hardware handles making the writes durable. Buffering also enables coalescing of writes which reduces the number of persists issued to NVM.

ASAP flushes all persists automatically, as soon as possible. Flushes could, however, arrive out-of-order at the memory controller, which could lead the system into an inconsistent state during crashes. ASAP handles this by storing *fixup* information in the memory controller, which is flushed on failure using ADR. The recovery information is used to resolve the inconsistency during a crash. ASAP tracks cross-thread dependencies using coherence, and resolves dependencies using new inter-thread messages.

A. Persistency model

ASAP builds on the `ofence` and `dfence` primitives from HOPS to enable applications to build high-level atomicity mechanisms. Writes within a thread are ordered using `ofence` while `dfence` guarantees that all earlier writes in that thread are persisted. `ofence` can be used to order log updates before data updates and `dfence` can be used at the end of transactions or after the completion of a data

structure update to ensure durability before responding to the client.

The key design aspects such as eager flushing, speculative updates to memory, and cross-thread dependency handling are not tied to the persistency model. We discuss and evaluate the design of ASAP that supports 1) *epoch persistency* or 2) *release persistency* [18]. The key difference between the two models is when cross-thread dependencies arise. With *epoch persistency*, a cross-thread dependency arises when a thread issues a conflicting access to any address which has been modified by another thread. With *release persistency*, a cross-thread dependency is detected only when a thread's `acquire` synchronizes with another thread's `release`, whether or not the address refers to persistent or volatile memory. Figure 4 shows the ordering semantics of the two persistency models. Persists within a thread are ordered using 2-sided persist barriers (`ofence`) while `acquire` and `release` act as 1-sided barriers.

B. Nomenclature

We define some of the terms useful for understanding the design of ASAP. *W*rites refer to stores issued to addresses in NVM. We use the term *flush* to indicate a write was sent to the memory controller, where ADR will ensure it is stored to NVM on a failure. We use the term *persist* to indicate that a write has reached the persistence domain. All flushes and persists occur at cache-line granularity.

- An **epoch** is a region of code separated by persist barriers within a thread. Writes in an epoch can be flushed concurrently. Writes in a later epoch should not survive a failure unless all writes from its preceding epochs also survive.
- **Epoch number** labels an epoch. Writes are tagged with the epoch number of the epoch it belongs to.
- An epoch is **safe** if all ordering constraints are satisfied, i.e., (1) all previous epochs on the same thread have been committed and (2) if it is dependent on an epoch from another thread and that epoch has committed.
- A **future epoch** is one that is not safe.
- An epoch **completes** when all the persists in the epoch have been flushed.
- An epoch **commits** when it is safe and is complete.

We use *thread* to refer to a CPU core that supports a single thread. We leave hyperthreading to future work.

C. Eager Flushing

ASAP employs eager flushing wherein it flushes writes in different epochs concurrently. ASAP speculates that writes from earlier epochs will eventually become durable and so it eagerly flushes writes from future epochs. This enables ASAP to overlap flushes from different epochs and use the total available system bandwidth more efficiently. ASAP differentiates between writes that are in the current flushing epoch and writes in future epochs. When ASAP flushes a persist in a future epoch, it marks the persist as ***early***. Eager flushing reduces latency and improves bandwidth by removing stalls needed to order epochs across multiple memory controllers, and to order dependent epochs on different threads.

D. Speculative updates to memory

When the memory controller receives an *early* flush, it speculatively persists the write in memory. The system might crash after the update (mis-speculation) and leave memory in an inconsistent state. To handle this, the memory controller saves recovery information in the memory controller before speculatively persisting the write. Specifically, it creates an *undo* record for the speculatively updated address by reading the value at that address from memory before issuing a write. ASAP includes the recovery information in the ADR domain, which can be flushed to NVM on a power outage. If the system crashes, ASAP reverts the state of memory using the *undo* record in the memory controller.

E. Cross-thread dependencies

Similar to HOPS [6], ASAP extends the coherence protocol to track cross-thread dependencies. ASAP augments coherence messages with dependence information. When a thread receives a coherence request for a cache-line, it replies with the data and current epoch number and then starts a new epoch. The core requesting the cache-line starts a new epoch that is dependent on the remote thread's epoch. Note that with *release persistency*, dependence information is sent only if the coherence request is for a cache-line touched by an `acquire`/`release`. With *epoch persistency*, by creating new epochs on cross-thread dependencies, circular dependencies can be avoided. This approach is borrowed from the epoch deadlock avoidance mechanism proposed earlier [14]. Since ordering is cheap in ASAP, this has little effect on performance. Furthermore, it is always safe to split an epoch into several smaller epochs, as they ensure the same ordering requirement. Note that, with *release persistency*, regular coherence requests do not establish a dependency, so ASAP requires race-free code.

ASAP applies eager flushing to cross dependencies. ASAP allows writes from a dependent thread to be flushed before or concurrently with writes from a source thread. This lets the dependent thread drain its buffer eagerly without having to wait for the source thread to complete its epoch. Like before, the dependent thread marks the writes as *early* when it sends the packets to the memory controllers. Figure 1c illustrates how ASAP improves flushing concurrency by allowing dependent threads to flush eagerly.

Cross-thread dependencies are resolved using thread-to-thread communication. When an epoch commits at the source thread, it notifies the dependent thread about the commit by sending an explicit message. This direct inter-thread communication has three advantages: (1) it scales

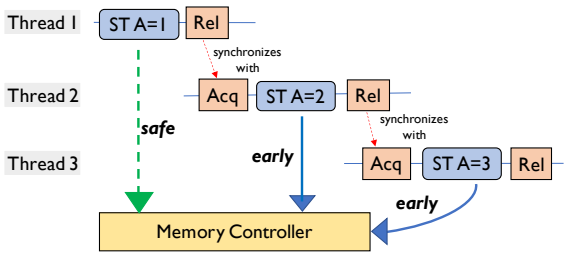


Figure 5: Write collision

better as there is no single point of contention unlike the global register in HOPS [6], (2) it avoids costly snooping mechanisms used in DPO [15], and (3) it reduces the resolution latency and saves energy compared to polling.

F. Write collision

Since ASAP flushes writes eagerly, writes to the same address can be flushed concurrently from multiple threads. A problem arises when multiple *early* flushes to the same address arrive at the memory controller out-of-order. For instance, a write with an older value could arrive at the memory controller after writes with newer values. Updating the memory speculatively and maintaining recovery information won't work as it could leave the memory with a stale value.

Consider the example in Figure 5. Suppose initially $A = 0$. All three threads write to A, but *early* flushes $A = 3$ from thread 3 arrives at the memory controller, followed by $A = 2$. When the MC speculatively updates memory with value $A = 3$, it stores an *undo* record of $A = 0$. When it speculatively persists $A = 2$, the value in memory is 3, so the MC stores the *undo* record $A = 3$. Memory now has the incorrect value due to the reordering of the two writes. Furthermore, If the system fails now, the correct value $A = 0$ has been lost, so ASAP cannot recover to a consistent state.

ASAP handles such write collision with additional record keeping. An MC creates *delay* record when an *early* write arrives and an *undo* record for that address already exists. A *delay* record contains the value of a write from an epoch that has not yet committed; once it is committed, the value from the *delay* record becomes the safe value and is copied into the *undo* record. In the above example, the MC speculatively updates memory with value $A = 3$ but creates a *delay* record for $A = 2$. The MC applies the delay record to update the undo value when the epoch containing $A = 2$ commits. Note that more than one *delay* record may be created if multiple *early* writes arrive at the memory controller.

V. ASAP Implementation

We implement ASAP for the x86-64 architecture to allow a comparison against Intel's currently available systems. The x86-64 ISA enforces TSO (Total Store Order) consistency model and lacks support for explicit acquire and release operations in the ISA. For *release persistency*, ASAP requires the programs to provide acquire/release information either through extensions to existing instructions or through annotations. We use acquire/release annotations in our programs.

A. Hardware Structures

We first describe the hardware structures used to implement ASAP; later, we describe the overall operations.

Persist Buffers (PBs): These are per-core circular buffers alongside the private caches similar to HOPS [6] and DPO [15]. Writes to NVM are buffered and tracked in the PB. When a core issues a store to an address in NVM, the write is simultaneously updated in the caches and enqueued in the PB. PBs flush writes in the background without the core's intervention. Memory is updated by flushing data from the PBs. Cache-lines for NVM evicted from the LLC are dropped and are not written back to memory.

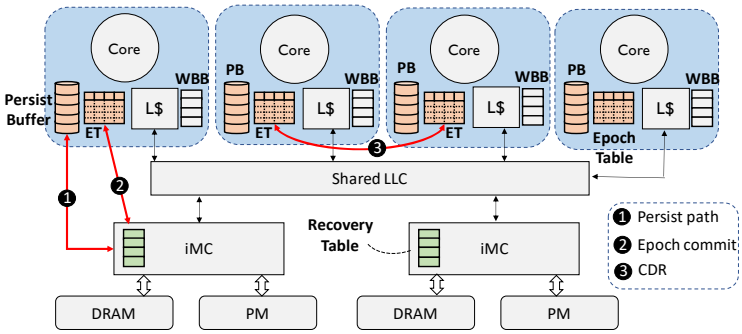
ASAP uses logical timestamps to label epochs. Each core has a timestamp register for the current active epoch. When an entry is added to the buffer, it is assigned the timestamp register's value. PBs also track the status of the flush. While flushing, PBs use the timestamp of the entry to look up in the Epoch Table (described below) to determine if a flush is *safe* or *early*. To notify the memory controller if a flush is *early*, PB sets a bit in the packet sent to the MC. PBs delete entries after they have been acknowledged by the MC.

Epoch Tables (ETs): The per-core epoch tables are CAMs that hold metadata about in-flight epochs. An epoch table resides next to the core's persist buffer. It tracks information about the status of in-flight epochs and cross-thread dependencies. For each ongoing epoch, the epoch table tracks the number of writes waiting to be flushed or awaiting acknowledgment. When a cross-thread dependency is detected via coherence message, the core records the source thread and epoch in the ET. The source thread adds the dependent thread to its epoch table as well.

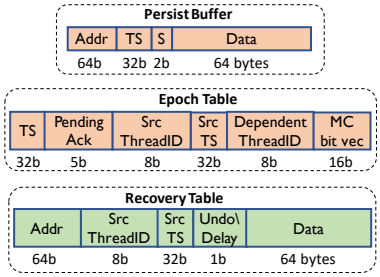
Execution of an `ofence` increments the current epoch timestamp and adds a new entry to the ET. When the core issues a `dfence` instruction, the ET waits to respond until all the in-flight epochs in the table have been committed. This ensures that all the writes prior to the `dfence` have been persisted. The core stalls until the ET responds.

Recovery Tables (RTs) are CAMs residing in the memory controller that hold *undo* and *delay* records. *Early* flushes could create either *undo* or *delay* records when they reach the MC. *Undo* and *delay* records store the address, data, threadID, and timestamp of the flush that created it.

Undo records are used to store the safe state for an address, i.e., value in memory prior to the speculative persist or the value written by the most recently *safe* flush. Undo records are created by reading the value from memory before speculatively updating it. Such a read-modify-write approach is acceptable because (1) NVM has read/write asymmetry, with read bandwidth much higher than write bandwidth, (2) only the number of reads increase (the number of writes



(a) System Overview. Colored structures are modifications in ASAP. Orange structures belong to volatile domain while green structures belong to persistence domain.



(b) Hardware implementation of Persist buffers (per core), Epoch Table (per core) and Recovery Table (per memory controller)

Figure 6: ASAP Implementation overview

Event	Action	
	Undo record for address *not* present	**Undo record for address present**
Safe flush arrives	Update memory	Update undo record
Early flush arrives	- Create undo record - Speculatively update memory	Create delay record

Table I: Handling of incoming flushes at the MCs

might decrease because of coalescing). (3) XPBuffer [1] in Intel Optane Persistent memory caches most recently accessed lines. Write would mostly hit in this cache.

Delay records hold updates that belong to epochs that have not yet been committed. The value in the *delay* record is the value flushed by the PB. *Delay* records are processed when the epoch they belong to commits (see Section V-C).

B. Handling incoming flushes

Eager flushing can reorder flushes to the memory controller. The flushes cannot be written out to memory naïvely. The memory controller re-orders some of the writes using *delay* records. Table I shows the different ways in which the memory controller handles incoming flushes. We briefly explain the actions here:

1) *Incoming **safe** flush **without** matching undo record*: This is the normal case where the flush updates the memory normally.
2) *Incoming **safe** flush **with** matching undo record*: Since an *undo* record exists, the address in memory has been speculatively updated with a later value. The older value in the incoming *safe* flush is not copied to memory to avoid losing the newer value already in memory, and instead is stored in the *undo* record.
3) *Incoming **early** flush **without** matching undo record*: The memory controller creates an *undo* record and speculatively updates memory.
4) *Incoming **early** flush **with** matching undo record*: Presence of an *undo* record implies memory is in a speculative state. The memory controller delays the incoming *early* flush by creating a *delay* record to process this flush later when its epoch commits.

C. Epoch commit and cross dependency resolution

ETs determines when an epoch is safe, complete, or can be committed. First, it recognizes an epoch E_n as safe when (i) E_n's preceding epoch has committed, and (ii) E_n has no cross dependency, or if it does, the ET has received a Cross Dependency Resolved (CDR) message from the source thread (described below). Second, ET recognizes an epoch as complete when the PB has received *ACK*s for all the writes in the epoch from the memory controller. ET commits an epoch when it is both safe and complete.

When an ET detects that an epoch can be committed, it sends *commit* messages to the memory controllers. In order to avoid unnecessary communication between the ET and the memory controllers, the ET records the memory controllers to which *early* flushes were issued.

A memory controller, on receiving a *commit* message, searches the RT and deletes any *undo* records belonging to that epoch, making space for new records. If any *delay* records exist for the epoch, they are processed as if the corresponding flushes just arrived at the MC (Section V-B): if no *undo* record exists, the *delay* value is persisted to NVM; otherwise the *undo* record is updated with the *delay* value.

After receiving an *ACK* for the *commit* message from the memory controllers, the ET considers the epoch to be committed. It then sends a *CDR* (Cross-thread Dependency Resolved) message to the dependent thread (if it exists). The dependent thread resolves the dependency (clears the metadata in its ET) when it receives the *CDR* message.

Unlike prior designs [15], [16], the recovery tables in the memory controller do not have to track epoch ordering. Instead, ETs track ordering information and send *commit* messages to the memory controller in the correct order. This strategy allows the recovery tables to commit writes in the right order without having to compare epoch timestamps.

D. Handling insufficient Recovery Table space

One of the challenges in ASAP is that RTs are finite. There may be no space in the RT to handle an incoming *early* flush. ASAP uses negative acknowledgments (*NACK*) to reject *early* flushes if there is no space in the RT.

Such a mechanism lets the memory controller exert back-pressure on the PBs. If a flush is *NACK*ed, PBs pause eager flushing and fall back to conservative flushing wherein only *safe* flushes are issued. These never allocate space in the RT. PBs waits until the epoch becomes safe and then retries the flush but this time as a *safe* flush instead of an *early* flush. Eager flushing resumes after the current epoch commits.

E. Crash handling

On failure, memory controllers are notified of a pending shutdown. MCs flush all writes in their WPQ (Write Pending Queue). Along with these writes, memory controllers write the values in the *undo* records to memory, thereby unwinding the effects of speculative updates. After this, no data from unsafe epochs remain in memory. *delay* records don't play any role here since they belong to epochs that were not completed before the crash and had not updated memory yet. Memory is restored to a consistent state during the crash and doesn't require additional recovery in hardware after restart.

F. Discussion

Handling private cache evictions. Cross-thread dependencies are established when a coherence request is forwarded to the thread that last wrote to it. It is possible that the cache-line is evicted from the private caches while preceding writes are still enqueued in the PB. Past designs [17], [18] solved this problem by managing the cache evictions in a small buffer. The cache-line eviction is delayed until all the preceding writes are persisted. ASAP uses a similar write-back buffer (WBB) [17] to prevent cache evictions before preceding writes are persisted. WBB records the tail index of the persist buffer when the cache initiates the eviction. The cache-line is written back from the WBB when the persist buffer flushes the corresponding index entry.

Handling early LLC cache-line evictions. In ASAP, cache-lines for persistent memory are dropped when they are evicted from the LLC since writes take the persist path through the persist buffer. A cache-line might be queued in the PB while it is evicted from the LLC. Loads to this address cannot read from memory since the latest value resides in the PBs and not in memory. However, these events are very rare. Some previous designs [28], [40] leverage the fact that writes in non-temporal paths almost always complete before the temporal counterparts. Nevertheless, this problem could arise in ASAP when flushes are *NACK*ed. When a flush is *NACK*ed, the data sits in the persist buffer until it is safe to be reissued. The cache-line pertaining to this flush might face an LLC eviction during this time. To overcome this, ASAP uses a counting Bloom filter at the memory controller, similar to HOPS [6]. The Bloom filter is populated with flush addresses that were *NACK*ed. ASAP uses the Bloom filter to check address of the cache-line being evicted from the LLC. If it hits, the cache eviction is delayed since the corresponding write in enqueued in the PB. When the write is retried, the address is removed from the Bloom filter and the cache-line is evicted.

CPU cores	4 cores, 8-way OoO, 2GHz
L1D caches	private, 32kB, 8-way, 1ns
L1I caches	private, 32kB, 8-way, 1ns
L2 cache	private, 2MB, 8-way, 10ns
LLC	shared, 16MB, 16-way
Coherence	MESI Three level
Memory controllers	16 entry WPQ, 32 entry RT
PM	Read = 175ns/Write = 90ns
Persist buffers	32 entry, flush = 60ns

Table II: Simulator configuration

Benchmark	Data structures	Description
Nstore		PM-native DBMS
Echo		Scalable key-value store
Vacation		Travel reservation system
Memcached		In-memory key-value store
ATLAS [30]	heap, queue, skip list	Insert/delete elements
CCEH [7]	extendible hashing	Insert/search elements
Fast_Fair [9]	B+-Trees	Insert/search/delete elements
Dash [8]	level hashing, extendible hashing	Insert elements
RECIPE [4]	radix tree, hash table, masstree	Insert elements

Table III: Workloads

DMA coherence. To ensure that DMA reads and writes are coherent with the CPU caches, software needs to issue `dfence` before initiating DMA operations. This ensures that all writes queued in the persist buffers are persisted.

Context switches. To ensure correct ordering when the operating system migrates a software thread from one core to another, the OS must issue a `dfence` instruction to ensure the thread's data have been safely persisted.

VI. Evaluation

We implemented and evaluated ASAP using full-system simulation on gem5 [41]. We simulated a modern multi-core system with 2 memory controllers similar to Intel Xeon processors. The hardware configuration is summarized in Table II. We modeled NVM characteristics based on a study on Intel Optane [38]. Each core has a 32-entry persist buffer and 32-entry epoch table along with the private caches. Each MC has a 32-entry buffer for the recovery table.

Table III shows the workloads used to evaluate the performance of ASAP. We chose workloads from a variety of sources with a variety of programming models, emphasizing multi-threaded workloads. We use 3 classes of applications:

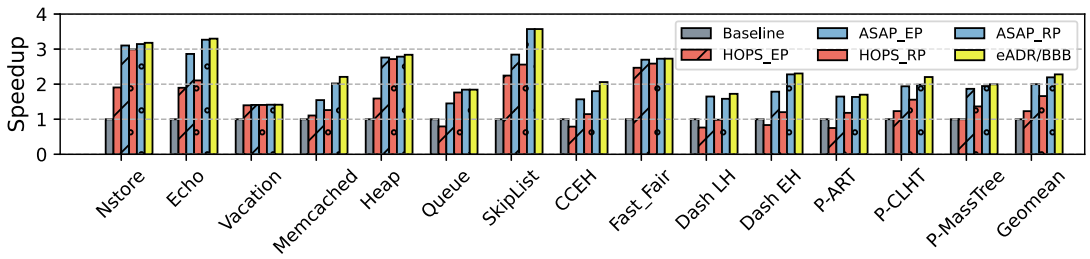


Figure 7: Performance study in a 4-core system

1) Benchmarks from the WHISPER suite with a mix of native (Nstore and Echo) and PMDK code (Vacation and Memcached) [6].
2) Hand-written data structures using the ATLAS persistence framework [30].
3) New concurrent persistent data structures, both hand-written (CCEH, Fast_Fair) and converted from DRAM structures (RECIPE, Dash).

We configure all applications to be update-intensive in order to stress PM write performance. For Nstore, Echo and Vacation we use default parameters used in WHISPER [6]. For the rest of the workloads, key and value sizes vary from 16B to 128B. Data is interleaved across memory controllers.

We compare the following designs in our evaluation:

- **Baseline**: This model replicates current Intel machines that support synchronous ordering through `clwb` and `sfence` instructions.
- **HOPS**: HOPS_EP [6] implements epoch persistency and HOPS_RP implements a variation of HOPS that provides release persistency. We make changes to the polling implementation in HOPS. The original implementation unrealistically polled every cycle and assumed read took a single cycle. We updated HOPS to poll every 500 cycles with each access of the global TS register taking 50 cycles.
- **ASAP**: ASAP_EP supports epoch persistency and ASAP_RP implements release persistency. These designs implement PBs, ETs and RTs to provide speculative persistence.
- **eADR/BBB**: This model implements a system with eADR. We also implemented an optimistic version of BBB [19]. BBB's performance is very close to that of a system with eADR. We therefore use a single graph to represent the performance of both eADR and BBB.

For all models, we assume ADR, i.e. the Write Pending Queues in the controllers are part of the persistence domain.

A. Performance study

Figure 7 compares the performance of all the models in a 4-core 2-MC system. Speedups are normalized to the Intel baseline. ASAP outperforms the baseline and HOPS while performing close to eADR/BBB for almost all workloads.

Comparison to baseline. As expected, the baseline model is the slowest as it stalls the CPU frequently waiting for cache flushes to complete. By decoupling ordering and durability, ASAP can overlap flushing with useful computation. ASAP_EP and ASAP_RP offers a speedup of $2.1\times$ and $2.29\times$ on average respectively. The speedup is significant in applications without frequent durability fences as it allows ASAP to make writes durable in the background without stalling the CPU. We see that ASAP performs well for applications with smaller critical sections. Vacation uses a coarse-grained lock while performing a query on the reservation system, and performs bookkeeping of volatile data before releasing the lock. By the time another thread acquires the lock, writes have been flushed out so early flushing is not beneficial.

Comparison to HOPS. ASAP outperforms HOPS for both persistency models. ASAP_EP improves performance by 37% on average over HOPS_EP while ASAP_RP improves by 23% on average over HOPS_RP. The performance improvement is significant for concurrent data structures such as CCEH, Dash and RECIPE. These workloads have smaller epochs with writes to different memory controllers. They also exhibit high cross-thread dependencies as shown in Figure 2, which leads to frequent flushing stalls in HOPS which ultimately results in core stalls because of insufficient buffering capacity. Instead of stalling, ASAP flushes writes early, making space in the persist buffers for newer writes. HOPS also stalls longer on `dfence` as it takes longer to drain the persist buffer completely. Having flushed writes early, ASAP has fewer durability stalls.

Comparison of persistency models. Both HOPS and ASAP perform better with *release persistency* than *epoch persistency*. This is mainly because the number of cross-thread dependencies is much higher with *epoch persistency* than *release persistency*. However, the difference between the performance is not significant with ASAP because ASAP is optimized to handle frequent cross-thread dependencies.

On the other hand, HOPS_EP's performance drops below baseline for concurrent data-structures such as queue, CCEH, DASH and P-ART. These applications have small epochs and frequent cross-thread dependencies. HOPS uses polling to resolve cross-thread dependencies, and if the polling period is longer than the time it takes to flush all writes in an epoch, HOPS stalls longer than baseline. For the rest of this section, we present results of models supporting

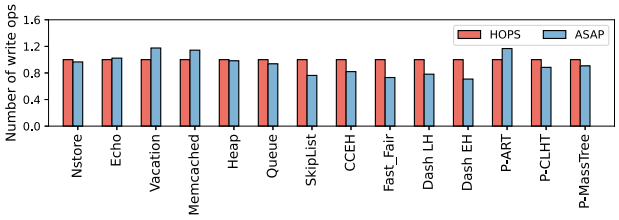


Figure 8: Number of PM write operations

release persistency as it performs better. We use HOPS to refer to HOPS_RP and ASAP to refer to ASAP_RP.

Comparison to eADR/BBB. ASAP_RP's performance is very close to eADR, on average within 3.9%. The primary cause of stalls in ASAP are `dfence` instructions used to ensure durability, and even then it rarely stalls because ASAP flushes writes early. Similar to BBB, cores could stall if persist buffers get filled up. This could happen if the rate at which the core issues writes is greater than the memory bandwidth. However, this doesn't happen often and as shown in Figure 10, the average PB occupancy is well below maximum capacity.

The downside of BBB is that on failure, it requires a larger battery to ensure that all the entries in the buffers are persisted along with any in-flight inter-core communication. ASAP can achieve comparable performance by using consolidated smaller buffers in the memory controller instead.

Write endurance PM has limited write endurance compared to DRAM. It is therefore important to reduce write traffic. Buffering enables coalescing which reduces the number of writes issued to memory and improves the write endurance.

Figure 8 shows the number of write operations in HOPS and ASAP normalized to HOPS. For most applications, ASAP has fewer write operations compared to HOPS. Some applications (Memcached, vacation, P_ART) benefit from additional coalescing in the persist buffer due to HOPS's conservative flushing. For most applications, however, ASAP achieves better coalescing for following reasons:

1) Latest value is already in memory: When writes arrive out of order, MCs often suppress the write issued from an earlier epoch that arrives later. Instead, the MC simply updates the *undo* record.
2) Coalescing in the Recovery Table: Flushes to the same address, belonging to the same epoch can be coalesced in the *delay* record in the RT.
3) Coalescing in the WPQ: For applications such as Dash-LH and Dash-EH, we observe that concurrent flushing from different threads can be coalesced in the WPQ (Write Pending Queue) in the memory controller.

Therefore, along with improving the performance, ASAP improves write endurance of NVM by improving coalescing. However, ASAP incurs additional cost in creating undo records before updating memory speculatively. On average, number of PM reads increases by 5.3% in ASAP over HOPS.

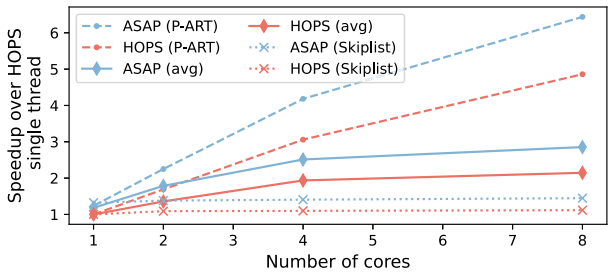


Figure 9: Sensitivity study of different number of cores

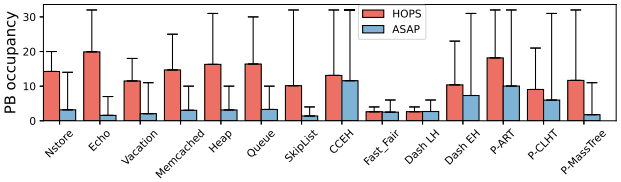


Figure 10: PB occupancy averages and $99th$ percentile. Bars plot the average, and the line caps show the $99th$ percentile.

B. Sensitivity studies

Number of cores. One of the key design goals for ASAP is to scale to larger servers. In that aspect, we evaluate the performance of ASAP and HOPS for *release persistency* by varying the number of cores. We varied the number of software threads in the applications accordingly. We fix the number of MCs to 2.

For a single thread, on average, ASAP improves performance by 18% over HOPS. With a single thread, there are no cross-thread dependencies. ASAP's improvement in throughput can be attributed to eager flushing, which allows concurrent flushes to both the MCs. ASAP is therefore able to utilize the system bandwidth better than HOPS.

ASAP scales better than HOPS as the number of cores increases. Due to limited space, we show the scalability for workloads that scale best (P-ART) and worst (Skiplist) along with the average in Figure 9. On average, ASAP achieves a speedup of 1.18×, 1.79×, 2.51× and 2.85× with 1, 2, 4 and 8 threads over a single thread of HOPS. HOPS is only able to achieve a speedup of 1.36×, 1.94× and 2.15× by increasing the threads to 2, 4 and 8. As the number of cores increases, the probability of cross-thread dependencies increases. HOPS falls off when the number of cores is increased because of its inefficiency in handling cross-thread dependencies. HOPS stalls frequently as it employs conservative flushing. ASAP on the other hand, avoids stalls as it employs eager flushing to flush writes from dependent threads early.

PB and RT occupancy Buffer sizes are crucial for two reasons: (1) it determines the cost (area, power) of implementing the design changes, and (2) it impacts the maximum performance achievable. ASAP adds 3 buffers in the form of PBs, ETs and RTs. ETs are very small since they neither

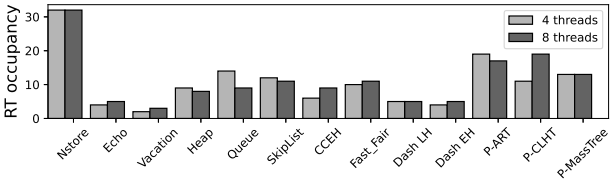


Figure 11: Recovery Table max occupancy

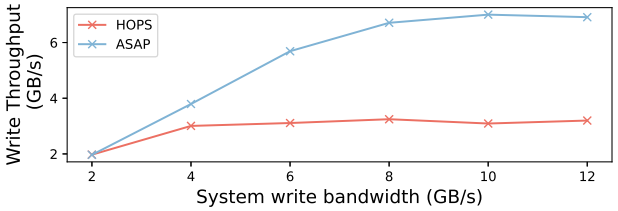


Figure 12: System bandwidth utilization

store addresses nor data. We study the occupancy of PBs and RTs to understand their impact on performance.

Figure 10 plots the average and 99^{th} percentile occupancies of the PBs in HOPS and ASAP. Since ASAP flushes writes eagerly, writes are queued for less time in the PB, leading to lower occupancy. Both the average and the 99^{th} percentile occupancy are much lower in ASAP. Although we simulate performance with 32 entry PB, we expect to observe similar performance with smaller PBs.

Figure 11 shows the maximum occupancy of the recovery table for both 4 threads and 8 threads. The max occupancy does not increase significantly from 4 threads to 8 threads. We believe that a small RT would improve performance even as applications scale. ASAP handles full RTs by falling back to conservative flushing. Therefore, ASAP's performance does not drop below that of HOPS even if the RT gets filled up. Nstore is an exception, in that it sometimes filled the RT and triggered *NACK*s to the persist buffer. However, *NACK*s did not hurt performance as the persist buffers were still able to flush data conservatively.

C. System bandwidth utilization

One of the advantages of eager flushing and speculative memory updates is the overlap of work across memory controllers. With eager flushing, ASAP can utilize more of the system bandwidth efficiently. To understand how well ASAP utilizes the available write bandwidth, we ran a custom bandwidth micro-benchmark. The benchmark issues 256-byte writes alternating across 2 MCs and the writes are ordered using an `ofence`. The results of the experiment are plotted in Figure 12. HOPS fails to utilize the system bandwidth efficiently while ASAP performs 2x better than HOPS on average owing to eager flushing which overlapped the writes to the 2 MCs.

D. Hardware cost analysis

We ran CACTI [42] simulations to study the area, access latency and energy overheads. At the time of writing this paper, CACTI supported 22nm node but nothing smaller. Table V summarizes the hardware cost for the persist buffer, epoch table and recovery table. We also present the numbers for a typical L1 cache for comparison. The sizes of the buffers were as specified earlier and the size of each field in the buffers is shown in Figure 6b.

Epoch Tables are small and do not add significant overhead. The benefits of having a persist buffer and recovery table outweigh the hardware cost they incur. Buffering improves performance significantly and enables coalescing which improves the write endurance. Recovery table enables speculative updates of memory which also speeds up the performance and reduces write operations.

Draining energy cost: Unlike BBB [19] and eADR [36], ASAP does not require a backup battery to flush data during crashes. ASAP extends the ADR domain in modern processors to store recovery information. With eADR, all the dirty cache blocks in the entire cache hierarchy needs to be flushed. Considering a server class CPU with 32 cores and cache sizes in Table II and assuming 50% of the cache blocks as dirty, about 42MB of data has to be flushed from the caches to NVM on power failure. BBB [19] reduces the amount of data to be flushed to about 64KB. ASAP requires less than 4KB of data to be flushed from the recovery tables in the memory controller. Additionally, ASAP needs to flush data only from the memory controllers and not the caches, therefore ASAP requires much less energy.

E. Qualitative comparison to other related work

Table IV summarizes ASAP's comparison to related work.
Comparison to LB++ [14]. LB++ augments the cache tag arrays to track ordering. Unlike ASAP, flushing begins only after an epoch is completed and all earlier epochs are complete. Since the design couples persist path to cache management, the system suffers from stalls on cache evictions and replacements. It also lacks durability guarantees required for ACID transactions and would suffer from long stalls even if it were to support it. We expect LB++'s performance to be lower than that of HOPS and ASAP.
Comparison to DPO [15]. Similar to HOPS and ASAP, DPO uses buffers alongside private caches to enqueue writes to NVM. DPO uses conservative flushing and stalls flushing on cross-thread dependencies. It uses broadcasts to resolve cross-thread dependencies which would be costly in a large system. Moreover, DPO does not support multiple memory-controllers. DPO's performance could be comparable to that of HOPS and lesser than that of ASAP.
Comparison to LRP [18]. LRP enforces release persistency by extending the cache tag array to include epoch numbers. LRP stalls on certain cache coherence state transitions. For instance, a forward request for a released cache-line could block until previous writes in the cache persist. ASAP instead records the dependency information and persists

	LB++ [14]	DPO [15]	HOPS [6]	LRP [18]	Strand-Weaver [17]	PMEM-Spec [20]	Vorpal [16]	BBB [19]	eADR	ASAP
Multi MC support	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Flushing Stalls[a]	High	High	High	High	Medium	None	High	None	None	Low[b]
Durability guarantees	No	No	Yes	No	Yes	No	No	Yes	Yes	Yes
Battery size	None	None	None	None	None	None	None	Medium	Large	None
Recovery Cost[c]	None	None	None	None	None	High	None	None	None	Low

Table IV: Comparison of related work.

[a]Flushing stalls refer to cycles when flushing is blocked to enforce ordering(Section III). [b]ASAP encounters flushing stalls only when RT is filled up (see Section V-D). [c]Additional recovery cost introduced by the hardware, excluding software recovery.

	Area (mm)	Access latency (ns)	Write energy (pJ)	Read energy (pJ)
Persist Buffer	0.093	0.402	30	28.876
Epoch Table	0.006	0.185	0.428	0.092
Recovery Table	0.097	0.413	31.5	31.5
32KB L1 cache	0.759	1.403	327.86	327.85

Table V: Hardware overheads of ASAP. Values are per-core for PB and ET, and per memory controller for RT.

writes speculatively without stalling. Hence, ASAP would perform better than LRP.

Comparison to StrandWeaver [17]. StrandWeaver is the only design that provides strand persistency. It performs better than HOPS as it allows epochs from different strands to be flushed concurrently. It uses *conservative* flushing to handle cross-strand dependencies. ASAP should outperform StrandWeaver as it allows flushing writes from different epochs concurrently including those dependent on other threads. ASAP could be integrated with StrandWeaver to support strand persistency and achieve higher performance.

Comparison to PMEM-Spec [20]. PMEM-Spec allows flushing speculatively any PM accesses without stalling or buffering. PMEM-Spec speculates that all accesses obey the ordering constraints and flushes them as they appear to the MC. Mis-speculations have high overhead as they are treated as failures and handled in software. For a single-MC system, PMEM-Spec performs similar to ASAP as it never stalls. But, in a multi-MC system out-of-order writes are common, leading to very high overhead from expensive recovery.

Comparison to Vorpal [16]. Vorpal is one of the few works that have addressed multi-memory controller systems. It uses distributed algorithms based on vector clocks to order persists across multiple memory controllers. Unlike ASAP, Vorpal delays the write until the memory controller deems it safe. As stated earlier in Section III, Vorpal incurs large overheads because of vector timestamps, and requires frequent communication amongst the memory controllers.

VII. Related Work

Related work on enforcing ordering was discussed previously, so we focus here on other closely related topics.

There has been growing interest in designing systems that provide software-transparent atomicity. These works [21]–[28], [43] provide hardware-assisted atomicity either through logging, out-of-place updates or hardware transactions. ThyNVM [44] provides software transparent crash consistency using a periodic hardware-assisted checkpointing mechanism. ASAP, in contrast, focuses only on ordering, which is component of many atomicity protocols. Themis [40] proposes lightweight extensions to the default x86 persistency model to provide ordering guarantees without explicitly requiring barriers but only supports programs that use undo-logging. LAD [45] provides atomically durable transactions in a multi-MC system. It uses buffers in the memory controllers to accumulate all updates within a transaction and uses a distributed 2PC-like protocol to atomically commit the transaction. ASAP reduces complexity by providing ordering instead of atomicity guarantees.

Recent research has also focused on developing data structures optimized for NVM [11], [46], [47]. FAST and FAIR [9] is a crash-consistent B-+tree, while CCEH [7] and Dash [8] implement persistent hash tables. Recipe [4], Pronto [48] and TIPS [12] are approaches to convert concurrent DRAM indexes to crash-consistent data structures with minimal changes. These structures do not rely on transactions but do benefit from ASAP's faster ordering.

VIII. Conclusion

NVM promises high-performance persistent data structures. Yet, the need to ensure ordering for consistency causes expensive stalls in current and proposed platforms, or limits the use of multiple memory controllers. Adding to that is the necessity for ordering persists across threads. ASAP relies on a novel early flushing mechanism that speculatively persists data out of order, and only ensures proper ordering on failure. ASAP maintains a small amount of recovery information in the memory controller to unroll the speculatively persisted data. This approach performs almost 23% better than past solutions to this problem, and within 3.9% of an ideal system.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful and useful feedback. We would also like to thank Mark Hill and Swapnil Haria for their support. This work is supported by National Science Foundation under grant NSF-CNS-1900758 and partially supported by the Parallel and Concurrent Computer Hardware and Software Research Fund generously provided by an alumnus.

REFERENCES

[1] I. Corp., "Intel optane persistent memory." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[2] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News*, 2011.

[3] 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," *ACM SIGARCH Computer Architecture News*, 2011.

[4] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: Converting concurrent dram indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[5] I. Corp., "Asynchronous DRAM Refresh." [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-overview.html

[6] 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 (ASPLOS)*. ACM, 2017.

[7] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[8] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: scalable hashing on persistent memory," *Proceedings of the VLDB Endowment*, 2020.

[9] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

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

[11] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "Wort: Write optimal radix tree for persistent memory storage systems," in *15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[12] R. M. Krishnan, W.-H. Kim, X. Fu, S. K. Monga, H. W. Lee, M. Jang, A. Mathew, and C. Min, "Tips: Making volatile index structures persistent with dram-nvmm tiering," in *2021 USENIX Annual Technical Conference (USENIX ATC)*, 2021.

[13] 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 (SOSP)*, 2009, pp. 133–146.

[14] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.

[15] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016.

[16] K. Korgaonkar, J. Izraelevitz, J. Zhao, and S. Swanson, "Vorpal: Vector clock ordering for large persistent memory systems," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, 2019.

[17] 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)*. IEEE, 2020.

[18] M. Dananjaya, V. Gavrielatos, A. Joshi, and V. Nagarajan, "Lazy release persistency," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[19] 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)*. IEEE, 2021.

[20] J. Jeong and C. Jung, "Pmem-spec: Persistent memory speculation," in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

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

[22] M. Cai, C. 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)*. IEEE, 2020.

[23] T. M. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018.

[24] 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)*. IEEE, 2017.

[25] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018.

[26] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016.

[27] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018.

[28] 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)*. IEEE, 2018, pp. 336–349.

[29] I. Corp., "Intel Persistent Memory Development Kit." [Online]. Available: https://github.com/pmem/pmdk

[30] 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 (OOPSLA*, 2014.

[31] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014.

[32] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017.

[33] A. Raad, J. Wickerson, G. Neiger, and V. Vafeiadis, "Persistency semantics of the intel-x86 architecture," *Proceedings of the ACM on Programming Languages*, 2019.

[34] S. Haria, M. D. Hill, and M. M. Swift, "Mod: Minimally ordered durable datastructures for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[35] P. Zardoshti, M. F. Spear, A. Vosoughi, and G. Swart, "Understanding and improving persistent transactions on optane™ DC memory," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020.

[36] I. Corp., "Persistent Memory Learn More Series." [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/pmem-learn-more-series-part-2.html

[37] S. Blanas, "From FLOPS to IOPS: The New Bottlenecks of Scientific Computing." [Online]. Available: https://www.sigarch.org/from-flops-to-iops-the-new-bottlenecks-of-scientific-computing

[38] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[39] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 496–508.

[40] S. M. Shahri, S. A. V. Ghahani, and A. Kolli, "(almost) fenceless persist ordering," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 539–554.

[41] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, 2011.

[42] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2017.

[43] K. Genç, M. D. Bond, and G. H. Xu, "Crafty: efficient, htm-compatible persistent transactions," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[44] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 672–685.

[45] 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 (MICRO)*, 2019.

[46] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," in *Proceedings of the VLDB Endowment*, 2015.

[47] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[48] A. Memaripour, J. Izraelevitz, and S. Swanson, "Pronto: Easy and fast persistence for volatile data structures," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

ARTIFACT APPENDIX

A. Abstract

This artifact includes the gem5 simulation code used to evaluate ASAP and other models (HOPS, baseline, ideal). Additionally, we provide the disk images (required for gem5 simulations) containing the workloads used to evaluate ASAP in this paper. We open-source this artifact to allow other researchers and developers to use and improve it in their own work.

This artifact includes the necessary scripts and tools to be able to reproduce the performance results shown in Figure 7. In this appendix, we briefly describe the necessary steps for compiling gem5 and executing the benchmarks.

B. Artifact check-list (meta-information)

- **Program:** gem5
- **Compilation:** GCC version 7.0.0 or above, Scons 3.0 or greater.
- **Models:** A total of 6 models including baseline, HOPS_EP, HOPS_RP, ASAP_EP, ASAP_RP and ideal.
- **Execution:** Scripts included in the artifact.
- **Metrics:** Execution time.
- **Output:** gem5 simulation stats are recorded for each run. The execution time for each simulation is obtained from these stat files.
- **Experiments:** Performance evaluation of 6 different models under 15 workloads.
- **How much disk space required?:** 50GB.
- **How much time is needed to prepare workflow?:** 1 hour (includes time to download images and compile)
- **How much time is needed to complete experiments?:** 2 days, running experiments in parallel is recommended.
- **Publicly available:** Yes, open-sourced on GitHub and archived on Zenodo.
- **Code licenses (if publicly available):** Same as gem5 (mostly BSD and MIT open-source licenses)
- **Workflow framework:** Custom workflow. Clone git, compile gem5, use scripts to run simulations, repeat these steps for all models, run script to gather results from the simulations.
- **Archived:** https://doi.org/10.5281/zenodo.5776777.

C. Description

1) How to access: The artifact is publicly available on Zenodo at https://doi.org/10.5281/zenodo.5776777. A GitHub repository for the same can be found at https://github.com/multifacet/ASAP. Supporting files such as disk and kernel images used with gem5 are included with the archive. The images can also be accessed at https://pages.cs.wisc.edu/~sujayyadalam/asap/. All the workloads used for evaluation are included in these disk images.

2) Hardware dependencies: gem5 is largely agnostic about the hardware it runs on. Here's a list of hardware requirements:

- 64-bit platform (tested on x86_64, gem5 supports multiple archs)
- Each experiment requires a little over 10GB of memory. The number of experiments that can be run in parallel is limited by the amount of memory available.
- 50GB of disk space to store disk images.
- Internet connection for cloning the repository and downloading the disk images.

3) Software dependencies: The version of gem5 used for implementing ASAP is tested on Ubuntu 18.04 and Ubuntu 20.04. Some of the software dependencies include:

- gcc version between 7 and 10 for compiling gem5.
- scons $\geq$ 3.0 for gem5 build environment.
- Python 2.7 (do not use Python 3 as it causes issues with gem5 version 20.0.0.3).
- latest versions of the following packages required for compiling and running gem5: `build-essential m4 zlib1g zlib1g-dev libprotobuf-dev libgoogle-perftools-dev libprotoc-dev libboost-all-dev pkg-config`.
- Python 3 for plotting graphs.

4) Data sets: Each workload used for evaluation uses its own data sets. Some workloads generate data sets before beginning execution and some use pre-generated data sets. The scripts included in the artifact ensure that the data sets are generated before the workload begins execution.

5) Models: Along with ASAP, this artifact includes 3 models: baseline, HOPS and ideal. Each of these models are implemented in gem5 and are available on separate branches in the GitHub repository. For ASAP and HOPS, there is an additional runtime parameter that is used to decide the persistency model (either epoch persistency or release persistency) to be used for the simulations.

6) About gem5 simulations: gem5 offers various CPU models with different features. In our experiments, we make use of 3 CPU models namely *x86KvmCPU*, *TimingSimpleCPU* and *O3CPU* along with the Ruby memory model. Each simulation has 4 stages:

Linux boot: *x86KvmCPU* is used the first phase which involves booting the kernel. x86KvmCPU uses KVM to accelerate the booting within the simulations.

Warmup: The linux boot phase is followed by the warmup phase where the *TimingSimpleCPU* model is used.

ROI: Once the warmup is complete, the detailed *O3CPU* model is used to simulate the region of interest or ROI.

Cool down: After the ROI is complete, the simulation is ended with a *TimingSimpleCPU* model.

The stats for the warmup, ROI and cool down phases are captured separately but written out to the same output file.

D. Installation

The `README.md` file of the https://github.com/multifacet/ASAP repository contains a detailed step-by-step installation guide. Summarizing the installation steps here:

- Install software dependencies.

```
$ apt install build-essential git m4
scons zlib1g zlib1g-dev libprotobuf-
dev python python-dev protobuf-
compiler libgoogle-perftools-dev
```

libprotoc-dev libboost-all-dev pkg-config

- Clone the repository.
  ```
  $ git clone https://github.com/multifacet/ASAP
  ```
- Create a directory called `disks` and download the disk images into it. Download the kernel image to the gem5 directory. Disk and kernel images can either be downloaded from Zenodo at https://doi.org/10.5281/zenodo.5776777 or from https://pages.cs.wisc.edu/~sujayyadalam/asap.
  ```
  $ cd ASAP; wget https://pages.cs.wisc.edu/~sujayyadalam/asap/vmlinux_12
  $ mkdir disks; cd disks
  $ wget -r -np -nd -A "*.img" https://pages.cs.wisc.edu/~sujayyadalam/asap/images/
  ```
- Checkout the branch corresponding to the model you wish to simulate. There are 4 branches in the repository: ASAP (default), baseline, HOPS and ideal.
  ```
  $ git checkout <branch>
  ```
- After changing the model, it is recommended to delete the build folder before re-compiling the new model.
  ```
  $ rm -r build
  ```
- Compile the gem5 model.
  ```
  $ python2 $(which scons) build/X86/gem5.fast -j<threads>
  ```

E. Experiment workflow

We have included a script '`run.sh`' that can be used to execute a workload with any model. We have also included a '`run_all.sh`' that would execute all the workloads for a single model in parallel. However, running all the workloads in parallel would require a large number of cores (around 16) and large memory (150GB). Note that root access (`sudo`) is required to run the scripts. This is because the simulations use *x86KvmCpu* model which uses KVM to accelerate the kernel booting in gem5.

For ASAP and HOPS models (ASAP_EP, ASAP_RP, HOPS_EP, HOPS_RP), an additional parameter for the persistency model needs to be passed to the script.

```
$ ./run.sh <workload> <persistency model>
```

F. Evaluation and expected results

Use the '*run.sh*' or the '`run_all.sh`' to execute all the workloads for that model. After all the workloads finish execution, change the model and repeat the process, i.e. recompile and re-run the experiments. Once all the models have been evaluated, use '`reproduce_results.py`' to plot the speedups similar to Figure 7.

Each gem5 simulation generates an output '`stats.txt`' file that can be accessed under the `results` directory. Some stats of relevance and their descriptions are listed in Table VI. The `reproduce_results.py` extracts the runtimes from these output files and generates the graph similar to Figure 7.

Stats	Description
cyclesBlocked	Cycles for which PB is unable to flush
cyclesStalled	CPU stall cycles because of full PB
dfenceStalled	CPU stall cycles because of dfence
entriesInserted	Total number of writes enqueued in the PBs
interTEpochConflict	Number of cross-thread dependencies
totSpecWrites	Number of early flushes
totalUndo	Number of undo records created

Table VI: Relevant stats and their descriptions

G. Experiment customization

Those familiar with gem5 can modify the models to try out new ideas. It would just require making changes to the source, re-compiling and running the experiments.

Running new workloads with these models is possible but requires additional effort. New workloads can be added to the existing disk images or new ones. Change the disk image in the `run.sh` script accordingly. More importantly, the new workloads need to be instrumented to include `ofence` and `dfence` in place of flushes and fences. A new gem5 script would also be needed. You could use the scripts in the `scritps` directory as reference.

Other runtime parameters can be varied to perform sensitivity studies and analysis. Refer to the `configs/common/Options.py` file for a list of run-time parameters. Parameters such as the persist buffer size and PMEM read and write bandwidth can be varied.

H. Notes

There are multiple sources of randomness while running gem5 simulations. Therefore the execution times for a particular experiment can vary from one run to another. For better accuracy, it is recommended to average the execution times from multiple runs.

On Ubuntu 18.04, there is an issue with `tcmalloc` and some simulations might crash mid-way. This is an issue with the gem5 source and not with the modifications made by us.

While running on Ubuntu 20.04, some Python modules might have to be installed. You can do so by using the following commands:

```
$ wget https://bootstrap.pypa.io/pip/2.7/get-pip.py
$ python2 get-pip.py
$ python2 -m pip install six
```