ASAP: A Speculative Approach to Persistence

1. Motivation

Persistent memory (PM) enables recoverable applications that can preserve in-memory data structures across system reboots and power failures. Correct recovery of these applications require crash consistency which constraints the order of writes to persistent memory. While programs could update data in the correct order, the cache hierarchy in modern systems may re-order the writes to persistent memory. Developers could use `flush` and `fence` instructions in current systems to ensure that writes reach memory in the required order. However, these instructions incur long stalls in today's systems. Past studies [6] have shown that these long stalls can cause slowdowns upto $7\times$ in applications with frequent ordering.

2. Limitations of the State of the Art

Several proposals have aimed to reduce the high cost of ordering either by entrusting the caches to enforce ordering [5] or use buffering [1, 2, 3, 6, 8] or through speculation [4]. Most of these designs fall short when 1) ordering persists across threads and 2) ordering across multiple memory controllers.

Handling cross-thread dependencies. Crash consistency for multi-threaded workloads require ordering of writes across threads. WHISPER workload analysis [8] found that applications had few dependencies across threads (cross-thread dependencies), where one thread persists data recently accessed by another thread. We found that cross-thread dependencies are frequent in newer high-performance concurrency-aware data structures designed for PM [7, 9].

Assuming infrequent cross-thread dependencies, prior designs employ *conservative flushing* to handle such dependencies. They stall flushing in the dependent thread until the source thread's persists completes. Figure 1b depicts the problem, writes in epoch[1] $E_{1,1}$ of thread T1 are not flushed until source thread T2 completes flushing epoch $E_{2,0}$. For instance, the buffers in HOPS [8] are unable to flush their writes for about 26% of the time on average. Such frequent stalls cause the finite buffers to fill up and exert back-pressure on the core pipeline, ultimately stalling the processor.

Multi memory-controller systems. Having multiple memory controllers in a single processor is common for server-class processors such as Intel Xeon. Enforcing ordering requires ensuring that writes are ordered correctly across multiple controllers. Writes at one controller may have to wait for writes to arrive and complete at another controller.

Some of the previously proposed designs either do not do not support multiple memory controllers [4, 6] and the other designs that do support are inefficient [2, 3, 8]. These designs wait for writes in the current epoch to be persisted before flushing writes from later epochs to different controllers. Figure 1a illustrates this case where writes to MC 2 are not flushed until writes in the first epoch are persisted by MC 1. By using such *conservative flushing* strategies, these designs fail to fully utilize the available system bandwidth.

[1] An epoch is a group of writes that don't have ordering dependencies among themselves. However, writes in different epochs are ordered.

3. Design Overview

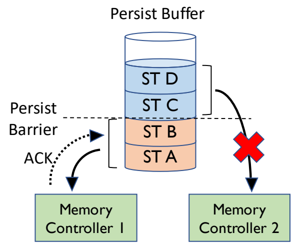
We propose ***ASAP*** which aims to improve performance of write ordering by avoiding flushing stalls in a multi memory controller system with frequent cross-thread dependencies. ASAP is a buffered persistency system that uses a separate data path for persisting writes to PM. Writes are queued in a hardware buffer called *persist buffer* and are flushed from this buffer to the memory controller.

Since crashes are rare compared to the frequency of persist ordering events, ASAP takes an *optimistic approach* to flushing writes. Unlike previous designs, buffers in ASAP flush all writes as soon as possible (ASAP) as shown in Figure 1c. We call this *eager flushing*. In turn, the memory controllers update memory speculatively with values from the eagerly flushed writes. Memory controllers maintain recover information to unroll the speculative updates in case of a crash.

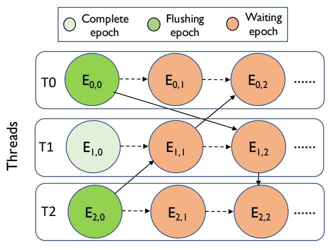
Eager flushing. ASAP speculates that writes from earlier epochs will eventually become durable, so it eagerly flushes writes from later epochs. ASAP differentiates between writes in current flushing epoch and writes in future epochs. When ASAP flushes a write in a future epoch, it marks the write as *early*. This way, ASAP overlaps flushes from different epochs going to different memory controllers and uses the total available write bandwidth more efficiently.

Speculative updates to memory. When a memory controller receives an *early* flush, it speculatively updates memory. The system might crash after a speculative update (mis-speculation) and leave memory in an inconsistent state. To handle this, ASAP 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 from memory before updating it. If the system crashes, ASAP reverts the state of memory using the *undo* record in the memory controller.

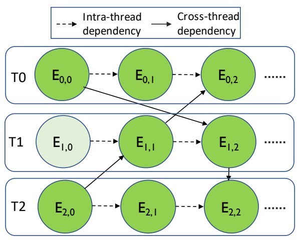
Crash handling. ASAP leverages Intel's Asynchronous DRAM Refresh (ADR) technology to save recovery information (*undo* records) in the memory controller. On failure, memory controllers are notified of a pending shutdown. Memory controllers write the values in the *undo* records to memory, thereby unwinding the effects of speculative updates. By undo-



(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

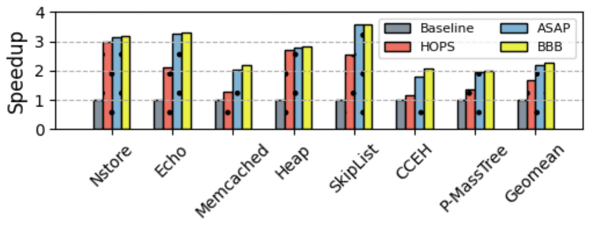


Figure 2: Performance study

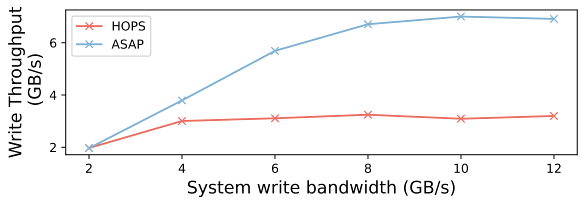


Figure 3: Bandwidth utilization

ing the speculative updates, memory is restored to a consistent recoverable state.

To support the above design, ASAP uses new structures: *persist buffers* to buffer writes going to PM, *epoch tables* to store metadata about on-going epochs and *recovery tables* in the memory controllers to store *undo* information.

4. Key Results

We implemented and evaluated ASAP using full-system simulations on gem5. We simulated a modern multi-processor with 2 memory controllers and modelled PM characteristics based on real world studies on Intel Optane. We evaluated ASAP on a wide range of benchmarks including workloads from the WHISPER [8] suite, data structures using ATLAS framework and highly concurrent data structures designed for PM.

Performance study. We compare the performance of ASAP to 1) Intel baseline, 2) HOPS [8], and 3) BBB [1] which has performance close to eADR. Figure 2 shows the performance comparison of a 4-core system. On average, ASAP offers a speedup of 2.3× over baseline. The baseline stalls frequently as the CPU waits for the cache flushes to complete.

ASAP improves performance by 22.8% on average over HOPS. The performance improvement over HOPS is significant for applications with high cross-thread dependencies as this leads to frequent flushing stalls in HOPS. Instead of stalling, ASAP flushes writes early, making space in the persist buffer for newer writes.

ASAP performs very close to a system with eADR, on average within 3.9%. Stalls are very rare in ASAP and they only occur when applications want to ensure durability after certain operations (durability fences). Unlike eADR and BBB [1], ASAP does not require a battery to backup the caches or the persist buffers. ASAP requires minimal data in the memory controller to be flushed in case of power failures.

System bandwidth utilization. Eager flushing and speculative memory updates enable better overlap of work across memory controllers. We ran a custom bandwidth micro-benchmark that issues writes separated by a fence and alternating across 2 MCs. The results of the experiment are plotted in Figure 3. HOPS fails to utilize the system bandwidth efficiently while ASAP performs 2x better than HOPS on average.

References

[1] 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.

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

[3] 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.

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

[5] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Efficient persist barriers for multicores. In *48th International Symposium on Microarchitecture (MICRO)*, 2015.

[6] 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.

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

[8] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with whisper. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017.

[9] 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.