Cache-Coherent Accelerators for Persistent Memory Crash Consistency

Ankit Bhardwaj
University of Utah

Todd Thornley
University of Utah

Vinita Pawar
University of Utah

Reto Achermann
University of British Columbia

Gerd Zellweger
VMware Research

Ryan Stutsman
University of Utah

ABSTRACT

Building persistent memory (PM) data structures is difficult because crashes interrupt operations, leaving data structures in an inconsistent state. Solving this requires augmenting code that modifies PM state to ensure that interrupted operations can be completed or undone. Today, this is done using careful, hand-crafted code, a compiler pass, or page faults. We propose a new, easy way to transform volatile data structure code to work with PM that uses a cache-coherent accelerator to do this augmentation, and we show that it may outperform existing approaches for building PM structures.

CCS CONCEPTS

• **Hardware → Memory and dense storage**.

KEYWORDS

persistent memory, cache-coherent accelerators

ACM Reference Format:
Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. 2022. Cache-Coherent Accelerators for Persistent Memory Crash Consistency. In *14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22), June 27–28, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3538643.3539752

1 INTRODUCTION

The availability of commodity persistent memory, or PM, (like Intel Optane DC Persistent Memory) has the potential to transform computer storage. By enabling direct CPU load and store access to persistent data structures, applications can interact with vast amounts of data in granular patterns while avoiding costly kernel boundary crossings, data movement, and serialization/deserialization overheads.

Hence, kernel file systems now map PM into processes to avoid overheads (e.g., Linux DAX), but this forces applications to handle crash consistency. If a process crashes while modifying a persistent data structure, its changes may be incomplete, and it may leave the data structure in an inconsistent state. Machines with PM support ensure the durability of dirty cache lines enqueued for write back to PM at the memory controller (ADR) and now even in CPU caches (eADR) [28]. However, this merely guarantees that dirty cache lines are persisted after a crash or power loss; it does not provide crash consistency guarantees. Processes must still ensure the crash consistency of their data structures at application level.

Many past schemes have been developed to provide crash consistency, but all existing approaches require interposing on stores to the persistent data structure, cutting into the direct-access benefit of PM. The standard approach is to rewrite code from scratch for crash consistency [4, 9, 11, 26, 30] by adding code that appends to a write-ahead log (WAL) before each store. On crash, the WAL is used to undo partially-applied operations to recover to a consistent state. This instrumentation can be automated by using a compiler to transform standard volatile data structures to support PM, but the injected logging code and ordering constraints still add overhead.

Hardware can also interpose on stores. Approaches that do this generally use page table protections to trigger write page faults on stores to track modifications [12, 15, 20]. This is a *black-box* approach in the sense that it can be used with unmodified code for volatile data structures. However, this approach suffers from extreme trap overheads on modern x86 CPUs (more than 1 µs per trap). It also suffers from high write amplification since it forces logging at a page granularity (4 KiB on x86) rather than at the specific size of the field being mutated in the persistent structure [1].

Our insight is that this interposition can be done today in hardware with low overhead without modifying host CPUs.

The idea is to use information exposed by CPUs to forthcoming cache-coherent accelerator devices (e.g., CXL [6]) to provide crash consistency for applications. In our approach, a process maps a physical address range exposed by a cache-coherent *persistence accelerator* device (or *PAX*) into its address space, and its threads interact with that region using normal loads and stores as if the region were the persistent structure itself (Figure 1). In turn, the device intercepts CPU requests for cache lines, and it proxies loads to PM. For stores, the device buffers modifications to cache lines, and it performs persistent undo logging before writing mutations back to PM. On crash, the structure on PM can be recovered to a consistent snapshot by rolling back partially-applied operations. To avoid the need for synchronous undo logging and write back after every operation on PM, operations asynchronously group commit at the device, ensuring processes need not block to wait for undo log entries to persist before continuing operation.

We outline how a PAX device would work and our plan to implement it on a cache-coherent FPGA [5]; we also describe a software-based prototype of the device that we have implemented. The device works together with a software library (`libpax`) to transform volatile data structures into linearizable and persistent data structures (e.g., C++ `std::unordered_map`) using the PAX to accelerate persistent snapshotting. PAX has several benefits over existing approaches to crash consistency:

Low Overhead. Loads and stores to PM operate out of CPU caches; cache misses are directly served by the device with no CPU traps and often from an on-device high-bandwidth memory (HBM) cache of PM. Undo logging for crash consistency is done by the device asynchronously for stores.

Black-Box Code Reuse. Existing volatile data structures can be transformed to be persistent without code changes.

Low Write Amplification. The device tracks modifications to the PM via cache-coherence messages, so it can log at cache line granularity, avoiding the high write amplification of page-based approaches for crash consistency.

No Working Set Size Limits. Unlike hardware transactional memory and approaches that buffer write sets in caches or DRAM [14], working set size is not limited by host- or device-side cache or volatile memory capacity. Dirty cache lines can be safely written back to PM to make room for new mutations even in the middle of operations on PM.

Efficient Use of PM Capacity. Despite snapshot semantics, only one copy of the persistent structure should be kept on PM; approaches [21, 22, 32] that create physical snapshots hurt the capacity cost of PM by 2× or more.

Why CXL, why now? Some cache-coherent accelerators already exist (e.g. Intel HARP [10], ETH Zurich's Enzian [5]), and augmenting CPU cache coherence to provide similar

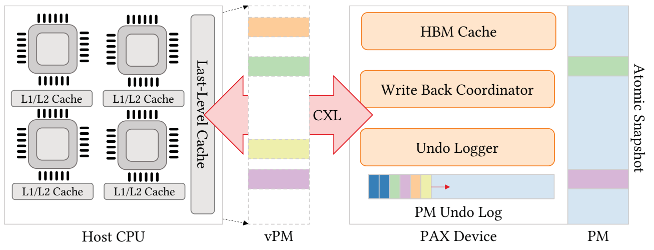

Figure 1: PAX Design Overview

crash-consistency guarantees have been explored in the past [8, 14, 23]. However, CXL-based accelerators are positioned to make PAX realizable soon and practical to use in real systems. This is because CXL eliminates the fragility of past approaches that required hardware changes [8, 14, 23, 27, 31] or that were tightly coupled to a specific CPU generation's microarchitecture and coherence details [2, 10] which made at-scale deployment impractical. CXL will allow PAX to be developed in such a way that it will work across *different CPU architectures and microarchitectures* without requiring changes to CPUs.

Recent work has begun to show the potential of these types of accelerators for disaggregated and distributed shared memory [1, 16] and VM migration [2]. Upcoming commodity CXL-enabled hardware will fuel at-scale deployment of new accelerator hardware and an explosion of use cases including PM applications. CXL devices are not yet available, but, as we will show (§4), it is possible to develop for CXL-based accelerators today without relying (solely) on software simulation.

2 BACKGROUND

To illustrate the problem of crash consistency, imagine a simple persistent hash table. When `put(key, value)` is called, several locations must be modified in the table: a key and value must be stored in a fresh allocation in PM, that allocation must be linked into the table, and the count of elements in the table may need to be updated. Regardless of complications like volatile caches and CPU store buffers, even if all written data is persisted, consistency will be violated after restarting if a crash occurs in the middle of these steps.

Write-ahead logging (WAL) is the standard approach to recovering from crash failures. It underlies persistent programming frameworks like Intel's PMDK, which provide PM structures like hash tables and more. WAL can use either redo or undo logging. In redo logging, structure operations log all locations and values to be updated; once the log entries persist, updates to the structure are applied. On a crash, missing updates are applied from the log. Similarly, in undo logging, the existing value stored in a persistent structure is logged for each location that must be modified. After a log entry recording the prior value persists, modifications

```
1 let mut allocator =
      HWSnapshotter<MyAllocator>::map_pool("./ht.pool");
2 let persistent_ht = Persistent<HashMap>::new(&allocator);
3 persistent_ht.insert(1, 100);
4 println!("Key 1 = {}", persistent_ht.get(1));
5 persistent_ht.insert(2, 200);
6 persistent_ht.persist();
```

Listing 1: Example of the PAX programming model in Rust.

are applied directly to the structure. The recovery procedure applies log entries to revert partially-completed operations.

Unfortunately, WAL-based approaches have drawbacks that cut into the benefits of PM's direct access model. The first is that code must be modified to interpose on updates to add logging, either by hand (like those provided by PMDK) or by using a compiler to inject code [3, 17]. Hence, creating correct, crash-safe persistent structures is a task left to experts, and existing volatile code is hard to recycle.

The second drawback is that logging adds overhead and many additional stalls to enforce the safe ordering of updates. In all forms of WAL, log entries must be ensured to be durable before the write back of changes to the structure begins, which requires costly `SFENCE` stalls. Without nuanced, structure-specific changes to code, stalls are incurred multiple times during a single logical operation like `put()` (log the allocation of a new key and value, `SFENCE`, write the new key and value, `SFENCE`, log the update of an internal pointer, `SFENCE`, update the internal pointer, `SFENCE`, etc.).

Our insight is that this interposition and these overheads can be offloaded to emerging cache-coherent accelerators.

3 DESIGN

In PAX users use unmodified code from standard volatile data structures to create persistent structures with low overhead. Here, we outline the PAX programming model, its handling of asynchronous logging and write backs, and its crash-consistent snapshotting and recovery procedures.

3.1 Programming Model

Listing 1 outlines the PAX programming model and shows how to use `libpax` to convert a (Rust) hash table into a persistent variant. `libpax` coordinates with a PAX device, which we plan to implement on an attached cache-coherent FPGA device (Figure 1). The PAX device interposes on accesses to the data structure, performs the necessary logging, and handles writing back updates to PM in a way that preserves crash consistency and snapshot semantics.

PAX Allocator Setup. Like other libraries, the application starts with a *pool* file that contains the persistent structure [11, 22], typically in a DAX-accessible file in PM (Line 1). `libpax` maps the corresponding vPM region into the address space of the process, and then it wraps the corresponding virtual address in an allocator object. The vPM addresses are marked as cacheable at the device (using CXL.cache semantics; see §4). All application accesses to the structure happen through those addresses, allowing the device to interpose on coherence messages for cache lines in vPM.

Data Structure Initialization. Then, the allocator is passed to a data structure constructor that accepts a custom allocator (many standard structure constructors do); here, the code is using an unmodified Rust hash table (Line 2). The allocator ensures all of the structure's allocations and accesses target the vPM region. (If the structure in the pool file needs recovery due to an earlier crash, `libpax` performs the needed recovery during this step; see §3.4.)

Loads. Line 4 calls the read-only `get()` method on the data structure. Loads that miss in CPU caches trigger the host CPU's cache home agent to forward a message to the device, which in turn fetches the corresponding cache line from the underlying PM and returns it to the CPU. Since vPM addresses are cacheable, future loads to the same line will hit in the CPU cache without communicating with the device.

Stores. Calls to `insert()` (Line 5) mutate state in the persistent hash table. On stores, the CPU's cache home agent contacts the device to request the cache line for modification. This gives the device a chance to perform undo logging, knowing that the CPU will soon produce a new value that must be written back to the structure at the requested address. To do this, the device fetches the old version of the cache line being modified from PM, and it logs the address and old value of the cache line in a persistent undo log.

Persisting. Finally, by calling `persist()` on Line 6 the library instructs the PAX device to persist a crash-consistent snapshot of the data structure.

3.2 Asynchronous Logging and Write Back

PAX asynchronously logs the old contents of a cache line to PM whenever a CPU asks to upgrade the cache line to exclusive mode in order to modify it. Rather than stalling the CPU while the device does logging, the device immediately acknowledges CPUs' upgrade request to modify cache lines without waiting for the logging to complete. This is safe since `libpax` only guarantees durability when a call to `persist()` completes.

Generally, the application issues `persist()` after a batch of operations, which works as a form of group commit [25]. CPU cores can read and modify cache lines without stalling for cache flushes or barriers for ordering and durability. Instead, the PAX device builds up a set of undo log entries that it flushes out asynchronously until `persist()`, making stores to PM nearly as efficient stores to non-crash-consistent structures. vPM is cacheable, so most operations are performed without consulting the device at all. Even for modified cache lines, the device is generally only informed the first time a cache line is modified after a call to `persist()`. Also, if desired, `libpax` can issue `persist()` periodically to limit undo log growth.

3.3 Crash-consistent Snapshotting

PAX guarantees that after recovery the application will always see vPM in a state that reflects the point in time of the last call to `persist()` executed. During recovery, PAX undoes unpersisted changes with the help of the log. If run in isolation, `persist()` ensures that the recovered PM is an *atomic snapshot*; it always appears to transition between successive persisted states atomically. We label each successive snapshot with an *epoch* number. The recovered state of the structure is always represented by the most recently persisted epoch.

One can think of the PAX as buffering all of the modified cache lines that should be atomically persisted when the next epoch finishes. However, solely using this approach has two problems: 1) the buffer could run out of capacity for modified cache lines, artificially limiting per-epoch working set size; and 2) the CPU may be caching modified cache lines that the device does not have in its buffer.

The device's asynchronous undo logging is key to solving both of these issues. The device-generated undo log is divided into epochs; if a crash occurs in the middle of writing changes to PM, the recovery process restores consistency by undoing all of the effects in the most recent (and not-yet-durable) epoch. This allows the device to freely modify PM during an epoch so long as it can undo partially-applied effects after a crash. So, the device can proactively write back a modified cache line to PM so long as its corresponding undo log entry is durable. This is easy to track since the undo log becomes durable at a monotonically increasing offset. Modified cache lines buffered at the device include the offset of their corresponding undo log entry, so the device knows when write back for that cache line is safe. This also avoids the capacity limitations like those that plague Intel's TSX hardware transactional memory [8, 19]; if the device is overwhelmed with modified cache lines that are part of the current epoch, it can still evict them and write them back once they are logged. In fact, the device buffer's eviction policy can try to minimize stalls by preferring to evict cache lines whose undo log entries are already durable.

Write back happens asynchronously as the application performs operations on vPM, even before `persist()` is called. Once `persist()` is called, the device ensures that all write back completes for all cache lines that the CPU modified during the epoch. This is done by iterating through each undo log entry as it persists and writing back any buffered new value to the corresponding cache line in PM.

However, one challenge is that a CPU may have modified some cache lines that it never evicted from its caches back to the device, which is the home of all vPM addresses. On a store that misses in the host CPU cache, the message from the CPU to the device only notifies the device that the CPU will modify the cache line, not what it will change it to. Hence, at end of an epoch, the device needs to ensure it has an up-to-date view of every cache line that the CPU could have modified. So, when `persist()` is called, the device iterates over every address in an undo log entry generated in the current epoch. For each address, the device triggers a CXL device-to-host message that is handled by the host CPU's memory controller requesting that cache line in shared mode, which both downgrades the cache line in all host CPU caches and causes the host CPU memory controller to forward the up-to-date value of the cache line to the device.

After this step and after all modified cache lines are safely written back to PM, the device writes the current epoch number to a special location in the structure's pool file. This write (once durable) atomically transitions the structure from the old epoch's snapshot to the new epoch's snapshot and `persist()` returns to the application.

3.4 Recovery

After a crash, the application reopens the same pool file and calls `Persistent<T>::new()`. `libpax` reads the epoch number stored in the pool, then it looks for undo log entries associated with the pool tagged with any later epoch number. For each such entry, `libpax` overwrites the corresponding cache line in PM with the value stored in the log entry. Next, it performs an `SFENCE`, and initializes the device and vPM as usual. Finally, it recovers the pool's allocator state, and it returns `Persistent<T>` which internally holds a pointer of type `T` to the persistent structure in vPM. From the application's perspective, there is no difference between constructing a new persistent map and recovering one; the application always recovers at the most recent persistent snapshot or with a new, empty instance of the structure.

3.5 Multi-threading

PAX requires the data structure code to be thread safe if multiple threads access the data structure concurrently; it does not provide concurrency control or transactions over vPM. Application code must ensure that `persist()` is only called when no thread is modifying the data structure, otherwise persisted snapshots may still include partial effects from ongoing operations.

4 IMPLEMENTATION & PROTOTYPING

We are currently implementing PAX. `libpax` is written in Rust; Rust's ownership semantics and borrow checker help statically enforce some of the safety properties of PAX. A C/C++ implementation is also straightforward, especially since C++'s STL structures accept custom memory allocators.

Our eventual goal is to implement a PAX device on a CXL 2.0-enabled FPGA where it will implement the CXL.cache protocol to interpose on coherence messages. This forces the host CPU cache home agents to forward snoop data and invalidate requests (`SnpData` and `SnpInv`, CXL 2.0 §3.2.4.3) in vPM. This lets PAX track which cache lines are being modified by host CPU cores.

On `persist()`, we plan to generate CXL device-to-host `RdShared` messages to force the host CPU to downgrade (and forward the current values of) its dirty cache lines before write back to PM. This is more efficient than forcing CPUs to issue `CLWB`s which are serialized, consume cycles, and cause complete evictions of cache lines and future cache misses (though future Intel CPUs promise to improve on this by simply downgrading cache lines to shared mode on `CLWB`).

CXL-enabled FPGAs are not yet available, so we are pursuing two alternative approaches today.

Cache-Coherent FPGAs. Other cache coherent accelerators can be used instead of a CXL accelerator like Intel's own discontinued HARP platform. For our hardware prototype, we are using an Enzian machine, a research computer with a Marvell Cavium ThunderX-1 48-core 2 GHz CPU connected to a Xilinx CVU9P FPGA via 24×10Gb/s lanes [5]. These lanes connect the ThunderX's cache-coherence bus to the FPGA, exposing the information we need to implement PAX.

The coherence messages observed by the FPGA are at a lower-level than what a CXL-enabled device would receive, and they are tightly coupled to the ThunderX's microarchitecture. To address this, our plan is to implement an "adapter" layer at the FPGA that filters and adapts the ThunderX's coherence messages to match the CXL specification so our implementation will be immediately portable to commodity machines when CXL devices arrive.

Enzian's CPU-to-FPGA coherence message latencies are higher than what are expected for CXL-attached device; we explore the impact of accelerator latency on expected performance in the next section.

Software-Simulated CXL Accelerators. Concurrently, we have also been working on a software-based, reference PAX implementation that runs on standard Intel CPUs. It uses a similar adapter layer to try to ensure that the software-based PAX still receives and reacts to CXL-defined messages.

To use this implementation, a process links against our PAX library as usual, but it is run via Intel's Pin [18]. Pin performs dynamic binary translation on the program, and it rewrites all loads and stores that target the vPM region. For each load or store, the rewritten code simulates a CPU cache; on a cache miss it sends a simulated CXL message to a PAX process over a shared memory queue. This CXL simulation layer is independent of PAX; it may be useful more generally for prototyping CXL-based systems.

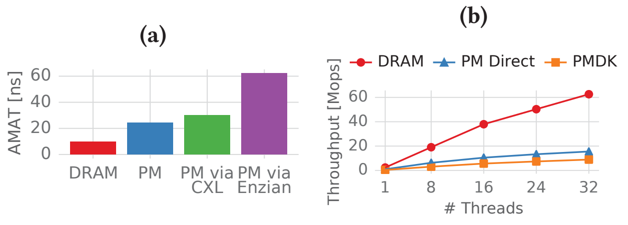

Figure 2: AMAT estimates and throughput benchmarks.

The software PAX runs on a separate host CPU core; it receives these simulated CXL messages and performs the same logic that we have described for the hardware-based PAX. Since communication is via shared memory, communication between program threads can be fast (easily 100 ns or less). Since persistent memory accesses take 305 ns [33], it is likely this PAX will be able to simulate realistic access times. The caveat to that is that Pin must instrument *every* store that accesses the vPM region, even if it would have been a CPU cache hit in the real implementation. Hence, we aren't yet sure how well it will perform overall.

5 PERFORMANCE

How will PAX perform compared to hand-crafted persistent memory structures (like those from PMDK) and compiler-transformed approaches? Though we don't have a hardware PAX implementation today to get a definitive answer, we can estimate it using a combination of benchmarks, latency measurements, and public information. Our estimates and measurements suggest that the performance of off-the-shelf data structures (`std::unordered_map`) using PAX may be similar to or better than hand-crafted PMDK data structures.

AMAT estimates. Our first analysis estimates how average memory access times (AMAT) would change for a simple hash table; it shows how much individual loads and stores would be slowed by adding a PAX between PM and an application. In the experiment, we measure cache miss rates at the L1, L2, and last-level cache running a standard hash table benchmark that performs `get()` operations on a single thread with small 8 B keys and values and a uniform random key access distribution on a Cloudlab c6420 [7] (`put()` latency is impacted similarly). We combine these miss rates with access latencies of each level estimated empirically using the same machine and publicly available information for Optane DCPMM [33], expected CXL latency [6], and Enzian coherence latency [5].

Figure 2a compares the estimated AMAT of different layers and media servicing last-level cache misses. DRAM and PM are not crash consistent; PM via CXL and PM via Enzian provide crash consistency. The key takeaway is that crash consistency for PM via a CXL-based PAX (blue bar) may only add 25% to application-experienced AMAT. This is because vPM is cacheable; though PAX interposes on PM access, the CPU caches eliminate most accesses to PAX. End-to-end application metrics (e.g., throughput) vary in sensitivity to AMAT between applications, but these results are promising as standard approaches to crash consistency would have additional stores and synchronous overheads that PAX eliminates.

Finally, our estimate for an Enzian-based PAX suggests that we can build a prototype PAX *today* that imposes about a 2× overhead over an eventual CXL-based implementation.

Throughput benchmark. Figure 2b compares the throughput of a volatile hash table (from Intel's TBB [13]) when it is placed in DRAM, PM directly (not crash consistent), and PMDK's TBB-based hash table for a write-only workload. For 32 cores, `PM Direct` performs ≈2× better than `PMDK` since `PMDK` writes to an undo log before updating the table.

We are optimistic that PAX will match or beat `PM Direct` for all workloads; hence, it may beat the hand-crafted PMDK hash table. This is because, though PAX does undo logging, its logging is asynchronous, so it provides nearly the performance of unsafe, direct PM access. By using high bandwidth memory (HBM) to cache at a PAX, it may be possible to beat direct PM performance and approach the performance of having the structure in volatile DRAM.

5.1 Bottlenecks and Optimizations

PCIe and PM Bandwidth. CXL is based on PCIExpress 5.0, so CXL-enabled accelerators could support up to 63 GB/s of full duplex bandwidth. A single CPU socket with an Optane DC PM DIMM per memory channel peaks at about 40 GB/s of read bandwidth and 14 GB/s for writes [33]. Workloads that reach these bandwidths would be rare, since most workloads will frequently hit in CPU caches. Overall, we expect that I/O bus bandwidth will not be a primary bottleneck in PAX.

Accelerator Bottlenecks. Host CPUs may collectively generate hundreds of millions of last-level cache misses per second that the PAX device must handle. In our initial Enzian-based prototype, we expect this to be a substantial bottleneck. The CVU9P FPGA that runs PAX is clocked at 300 MHz. To saturate the interconnect between the ThunderX-1 and the FPGA, PAX would need to respond to coherence messages on nearly every clock cycle. We plan to make PAX parallel and pipelined, but we expect this will still be a bottleneck. Hence, we expect that designs for other cache-coherent accelerators that include ASICs for handling coherence messages [10] would likely outperform our Enzian-based prototype.

Combining with Paging. Recent work suggests that paging-based approaches for tracking changes to remote memory suffer both in terms of performance and write amplification compared to a PAX-like approach [1]. However, paging may capture spatial locality well for some workloads. PAX must interpose on every last-level cache miss, but paging-based approaches only incur overhead on the first access to a page per epoch, which can be amortized for some applications. Our plan is to compare these approaches in detail for a variety of applications. We may find that a combination of the approaches works best. For example, it is possible for PAX to manage write backs to PM DIMMs attached to the host CPU memory controller. In such deployment, the application could directly map PM pages as read-only; on a write page fault, the page could remapped at read/write through addresses assigned to vPM, letting PAX track changes to the page at cache line granularity.

6 LOOKING FORWARD

PAX will provide near-native access times to PM structures with unmodified volatile data structure code with low write amplification and without hardware changes, stalls for logging, or working set restrictions.

Our work on PAX has already raised interesting questions that we are exploring. For example, we believe it may be possible to make `persist()` fully non-blocking, so that epochs overlap and threads never stall even during `persist()`; this is challenging since we cannot modify CPU caches to retain different cache line versions for epochs. Similarly, we are extending PAX to efficiently provide linearizability in a black-box fashion with highly concurrent workloads.

Integrating PAX with existing and future hardware is interesting since platforms have different capabilities; CXL.mem can support basic functionality, but it does not have as much visibility into coherence as CXL.cache, which has less visibility than Enzian [5]. Hence, it will be interesting to see what optimizations are possible with each approach.

Finally, different applications can use our techniques e.g., to enable efficient transactions within a cluster of machines by connecting FPGAs over a high-speed network or providing fault tolerance via remote memory [24, 29].

CXL will be here soon. Beyond coherence, CXL can give applications a new lens to view and interpose on their own operations. We believe PAX is exciting since it is an early step toward algorithms that benefit from exposing cache-coherence details directly to applications.

ACKNOWLEDGMENTS

We thank the reviewers for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking Software Runtimes for Disaggregated Memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[2] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. Project PBerry: FPGA Acceleration for Remote Memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 127–135, 2019.

[3] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 433–452, New York, NY, USA, 2014.

[4] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 105–118, New York, NY, USA, 2011.

[5] David Cock, Abishek Ramdas, Daniel Schwyn, Michael Giardino, Adam Turowski, Zhenhao He, Nora Hossle, Dario Korolija, Melissa Licciardello, Kristina Martsenko, Reto Achermann, Gustavo Alonso, and Timothy Roscoe. Enzian: An Open, General, CPU/FPGA Platform for Systems Software Research. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 434–451, New York, NY, USA, 2022.

[6] CXL 2.0 Specification. https://www.computeexpresslink.org/spec-landing.

[7] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[8] Pradeep Fernando, Irina Calciu, Jayneel Gandhi, Aasheesh Kolli, and Ada Gavrilovska. Persistence and Synchronization: Friends or Foes? *arXiv preprint arXiv:2012.15731*, 2020.

[9] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 859–872, 2020.

[10] Prabhat K Gupta. Accelerating Datacenter Workloads. In *26th International Conference on Field Programmable Logic and Applications (FPL)*, volume 2017, page 20, 2016.

[11] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 26th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[12] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 468–482, 2017.

[13] Advanced HPC Threading: Intel oneAPI Thread Building Blocks. https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html.

[14] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. DHTM: Durable Hardware Transactional Memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 452–465, 2018.

[15] Terence Kelly. Persistent Memory Programming on Conventional Hardware: The persistent memory style of programming can dramatically simplify application software. *Queue*, 17(4):1–20, 2019.

[16] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, and Ricardo Bianchini. First-generation Memory Disaggregation for Cloud Platforms. *arXiv preprint arXiv:2203.00241*, 2022.

[17] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[18] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 190–200. ACM, 2005.

[19] Darko Makreshanski, Justin Levandoski, and Ryan Stutsman. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. *Proceedings of the VLDB Endowment*, 8(11):1298–1309, July 2015.

[20] Leonardo Marmol, Mohammad Chowdhury, and Raju Rangaswami. LibPM: Simplifying application usage of persistent memory. *ACM Transactions on Storage (TOS)*, 14(4):1–18, 2018.

[21] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 499–512, 2017.

[22] Amirsaman Memaripour, Joseph Izraelevitz, and Steven 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*, pages 789–806, 2020.

[23] Tri M Nguyen and David Wentzlaff. PiCL: A software-transparent, persistent cache log for nonvolatile main memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 507–519. IEEE, 2018.

[24] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3), aug 2015.

[25] Steven Pelley, Thomas F Wenisch, Brian T Gold, and Bill Bridge. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment*, 7(2):121–132, 2013.

[26] Persistent Memory Devlopment Kit. https://pmem.io/pmdk/.

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

[28] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide For Developers*. Springer Nature, 2020.

[29] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, 2013.

[30] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 91–104, New York, NY, USA, 2011.

[31] Ziqi Wang, Chul-Hwan Choo, Michael A Kozuch, Todd C Mowry, Gennady Pekhimenko, Vivek Seshadri, and Dimitrios Skarlatos. NVOverlay: Enabling Efficient and Scalable High-Frequency Snapshotting to NVM. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 498–511. IEEE, 2021.

[32] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 623–637, 2020.

[33] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 169–182. USENIX Association, 2020.