PLayer: Expanding Coherence Protocol Stack with a Persistence Layer

Richard Braun
ETH Zurich, Switzerland
braunr@student.ethz.ch

Abishek Ramdas
ETH Zurich, Switzerland
abishek.ramdas@inf.ethz.ch

Michal Friedman
ETH Zurich, Switzerland
michal.friedman@inf.ethz.ch

Gustavo Alonso
ETH Zurich, Switzerland
alonso@inf.ethz.ch

Abstract

Mechanisms to explicitly manage data persistence for non-volatile main memories are fundamental for the correctness and performance of modern systems. So far, however, most solutions are primarily based on software techniques. In this paper, we design a persistence layer on hardware, to support correct handling of persistent lock-free data structures. By exploiting cache-coherence messages, persistence can be transparently managed by the hardware, with minimal user intervention. We have experimented with a partial design on a Soft-CPU running on an FPGA to explore the idea, and plan to further extend it into a real hardware implementation.

ACM Reference Format:
Richard Braun, Abishek Ramdas, Michal Friedman, and Gustavo Alonso. 2023. PLayer: Expanding Coherence Protocol Stack with a Persistence Layer. In *1st Workshop on Disruptive Memory Systems (DIMES '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3609308.3625270

1 Introduction

With the commercial availability of Non-Volatile Main Memory (NVMM), high-performance, persistent byte-addressable memory became a reality [3, 4, 36]. Upon a crash, NVMM data will remain but CPU registers and caches are still volatile and lose their contents when a crash occurs. Therefore, writing applications that provide recoverability following a crash is still an involved and time-consuming process. As a result, many paradigms for managing persistence in software have emerged [7–10, 13, 15–19, 21–30, 33–35, 37–39, 42–45].

ACM ISBN 979-8-4007-0300-3/23/10…$15.00
https://doi.org/10.1145/3609308.3625270

One of these paradigms relies on persistent snapshots and transactions. These are common approaches for providing persistence [9, 13, 15, 23, 24, 27, 29, 33, 34, 44], and are widely applicable. These approaches usually allow the programmer to augment their code with instructions on where to take a persistent snapshot, or which sections need to execute with all-or-nothing semantics. Then, they guarantee a consistent state from which the application can recover after a crash. A significant downside of these very general paradigms is that they usually come with high-performance degradation.

While the proposed paradigms for managing persistence in software already make the development of persistent applications easier, they often require modifications to the application code. To mitigate the overhead and code modifications, research on implementing persistence management at the hardware level has emerged [6]. The authors propose using cache-coherent accelerators to implement write-ahead logging in hardware on a single thread. This follows the trend of computer systems becoming more heterogeneous, to better adapt to today's data-heavy workloads. These systems use different purpose-built ASICs or FPGAs for offloading and accelerating specific tasks. This trend is reflected in recent standardization efforts, such as CXL, to provide standardized hardware interfaces and protocols for accelerator devices [2].

Nonetheless, different applications have different requirements when it comes to what data needs to be saved to allow recovery after a crash. For many applications, it is enough to persist certain critical data structures. In this context, lock-free data structures are a natural fit for persistence. That is because every operation on the data structure is guaranteed to leave it in a consistent state. Hence, as long as operations are persisted in the correct order, every state of the data structure is easily recoverable from [16, 18, 23, 31].

Lock-freedom guarantees are usually achieved through atomic Read-Modify-Write instructions. In this paper, we propose using the specific cache-coherence messages occurring during these atomic instructions, for implementing a persistence layer (PLayer) applicable to lock-free data structures, on hardware. This layer makes it easy for programmers to make their data structures persistent with minimal changes.

The Read-Modify-Write (RMW) cache-coherence messages allow the cache-coherent accelerator device to track memory operations on the host, and guarantee their persistence while maintaining a consistent state. Moreover, it can distinguish between writes that occur due to thread-local data manipulations, and updates to the shared data-structure state. Since thread-local data manipulations are not part of the data structure state, they do not require immediate persistence, which reduces the overhead caused by persisting data. Only when data becomes shared, we need to guarantee its persistence before any other thread relies on this data.

Our proposed construction is intended to work with upcoming standardized and commercially available cache-coherent accelerator platforms, such as those implementing CXL [2]. As these platforms are not yet available, we used Enzian [11], which provides a cache-coherent interconnect that allows tracking and manipulating coherence messages. This way, we evaluated most of our construction requirements against what current, real-world cache-coherence protocols offer. We experimented with a very early partial design on a Soft-CPU running on an FPGA to explore the idea and plan to further extend it into a real hardware implementation.

2 Background

2.1 Non-Volatile memory and persistent programs

Running an arbitrary piece of software on persistent memory alone is, unfortunately, not enough to guarantee recoverability after a crash since caches are still volatile. Since their contents are lost upon a crash, programs can remain in an inconsistent state. To overcome their volatility, one needs to write back every memory access to the NVMM, before the next operation is executed [23]. Other solutions try to reduce the expensive writeback overhead, but their performance is still degraded [7, 15–17, 33, 38]. To reduce their performance penalty, specialized hand-tuned data structures are used [8, 16, 18, 28, 31, 45]. These offer superior performance as many instructions can be postponed to recovery time if one can guarantee that the state after the crash remains consistent. Writing optimized software solutions for persistent memory, however, is not an easy task. The design of such optimized data structures is considered hard and error-prone [25], which is why automated transformations are valuable [23, 25, 30, 40]. Some general transformations also restrict their applicability to lock-free data structures as in lock-freedom, every operation leaves the data structure in a consistent state, which makes it an excellent fit for persistence transformations [17, 19].

2.2 Lock-Freedom and Read-Modify-Write instructions

In lock-free programs, system-wide progress is guaranteed at all times. Therefore, any thread can crash at any time, without creating a deadlock [20]. If a thread crashes unexpectedly, other threads must know how to deal with the data structure's state, after the interrupted operation's updates of the thread that crashed. As a consequence, lock-free data structures always keep memory in a consistent state [20]. This property is especially interesting in persistent algorithms, as it helps recovery after a crash. A common category of primitives often used in lock-free programs are Read-Modify-Write (RMW) instructions. RMWs are atomic instructions that modify and return the contents of a specified memory address atomically. RMWs are used to implement a variety of synchronization primitives. In lock-free data structures, they provide the mechanism by which threads can atomically modify a data structure, such that before and after the modification, a consistent state is reached. On a lower level, cache coherence messages are one of the mechanisms responsible for making RMW instructions work properly in directory-based multi-processor systems.

2.3 Cache coherence messages

Cache coherence is a prominent concern in multi-processor systems, where cores can store and modify copies of shared data in their private cache. These copies of data shared are kept coherent with each other using a coherence protocol. Coherence controllers such as the cache and memory controllers maintain cache line states and exchange messages to maintain coherence. Coherence protocols are typically named after cache lines states. For example, the MSI protocol allows cache lines to be either in Modified (M) state in one cache, Shared (S) across multiple caches or Invalid (I) when not present in a cache. These protocols can have additional states, e.g, MOESI with Owned (O) and Exclusive (E) states, MESIF with Forward (F) state, etc.

Coherence protocols are classified into directory-based and snooping protocols. Snooping protocols rely on a shared bus for maintaining coherence which comes with scalability issues. Directory-based protocols are designed for scalability and are used to achieve coherence across different nodes in a NUMA system. In directory-based coherence protocols, each node has its own directory that offers a central view of the states of the lines that belong to it. Consequently, the controller within every node can effectively determine the necessary messages to send for any given request, ensuring the requested cache line is coherent throughout the system. These messages serve to update cache-line states and exchange relevant cache-line data among the nodes. The protocol is usually proprietary leading to variations in functionality and performance among different vendors and CPU models. It is not limited to only be implemented on CPUs.

2.4 Cache coherent accelerator platforms

In recent years, systems have become more heterogeneous by incorporating accelerator devices like GPUs, FPGAs and other devices to speed up specific workloads. Traditionally, these devices were often connected through a peripheral bus

Algorithm 1. Persistent enqueue example. The red instructions are not needed with PLayer.

```
void enq(T value)
   Node* node = new Node(value);
   BARRIER(node);
   while (true)
      Node* last = tail.load();
      Node* next = last->next.load();
      if (last == tail.load())
         if (next == nullptr)
            if (last->next.CAS(next, node))
               BARRIER(&last->next);
               tail.CAS(last, node); return;
         else
            BARRIER(&last->next);
            tail.CAS(last, next);
```

such as PCIe. Nevertheless, another approach that started gaining traction recently, is to connect accelerators in a cache-coherent manner. Many approaches exist to connect accelerators coherently, including CXL [2], MPSoCs such as Xilinx's Zynq [41], etc. Some of these options offer *symmetric protocols* which allow both CPUs and accelerators to track cache lines ownership and send cache control messages. As a result, the accelerator can manage its own memory, and generate coherence messages as opposed to being controlled by the CPU. This opens up the possibility of many different algorithms, and therefore, we will focus on symmetrical protocols.

2.4.1 CXL.
CXL is an industry standard for connecting CPUs to accelerators, IO-devices and memory [2], which has been recently announced. For accelerator devices, it allows coherent memory access between the CPU and the accelerator device. In particular, CXL 3.0 is a symmetric protocol, which makes it suitable for implementing hardware-accelerated memory management for persistence and recoverability, without relying on software support. CXL 3.0, however, is still not commercially available.

2.4.2 Enzian.
Enzian [11] is a research computer developed by the Systems Group at ETH Zurich. It features a ThunderX CPU with 48 ARMv8.1 cores and a Xilinx CVU9P FPGA. The CPU and the FPGA are connected through the native cache coherent interconnect of the ThunderX CPU, utilizing the Enzian Coherent Interconnect (ECI) protocol, which is mostly based on the MOESI-protocol. Enzian is a symmetric coherent platform where each node is responsible for maintaining coherence of the memory attached to it. Enzian's FPGA maintains a directory controller (DC) which implements the coherence protocol and allows user logic on FPGA to directly access CPU memory, change cache lines' states, trigger inter-processor interrupts, etc. The DC on the FPGA exposes an AXI (Advanced eXtensible Interface) interface to access the FPGA attached memory. It also provides a simplified *local* (LCL) request-acknowledge interface for user logic on the FPGA to interact with the coherence protocol. Through this interface, user logic on the FPGA can clean and invalidate FPGA-homed cache lines that are cached in the CPU's LLC. As a result, user logic does not have to keep track of cache line states and can implement a much simpler protocol, while the DC maintains coherence.

These capabilities make this system ideal for exploring the possibilities of using cache-coherent accelerators for implementing persistence in hardware.

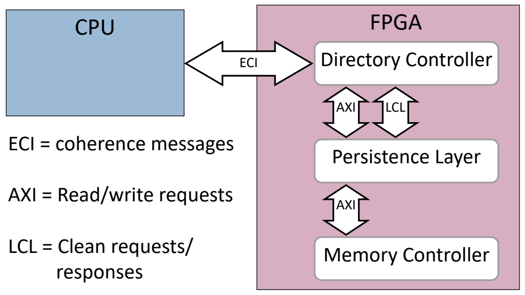

Figure 2. PLayer architecture. PLayer interacts with the DC to orchestrate CPU's access to memory and transparently provide persistence guarantees to software on CPU.

3 PLayer Design

PLayer guarantees persistence for lock-free data structures, by ensuring a consistent state [5, 23], using the hardware with minimal software support. Figure 2 shows the system's overview. We place the program's memory on the FPGA side, such that the FPGA manages its persistence. Whenever the CPU's last-level cache (LLC) wants to access a cache line homed on the FPGA, it sends a coherence message to the FPGA's DC through ECI. For most applications, the DC would be connected directly to the FPGA-attached memory and provide coherent access. We introduce a persistence layer that is located between the DC and the memory controller on the FPGA. This layer can observe CPU upgrade requests and orchestrate access to FPGA memory (AXI). It can also interact with the coherence protocol through the DC's local request-acknowledge interface by issuing clean and invalidate requests for FPGA lines cached in the CPU's LLC.

Algorithm 1 presents a code snippet of an enqueue operation of a durable lock-free queue [18]. The BARRIERs in red represent the explicit writebacks and fences the user must invoke in normal execution. However, with the existence of PLayer, these barriers (and reasoning about their correctness), can be omitted, since they are managed by PLayer.

We explain the PLayer protocol design through an example execution of a simple durable lock-free queue [18].

3.1 Initialization

First, the persistent data-structure needs to load PLayer and then allocate the data structure on the FPGA address space. Currently, this is done by *mmaping* a region of memory in the FPGA address space. Whenever the CPU accesses this region, the FPGA would receive necessary coherence messages from the CPU's LLC. We expect the platform [11] to allow explicit memory allocation/deallocation in the future.

3.2 Local node preparation

Before a thread inserts a new node into the data structure, it prepares it locally (Algorithm 1, line 2). These node manipulations are done by ordinary write instructions in the CPU's cache. At this point, the node is not accessible by other threads and this data does not have to be persisted yet. Eventually, the thread will issue a read-modify-write (RMW) instruction to atomically modify the pointer of the last node in the lock-free queue to point to the new node (Algorithm 1, line 9). As soon as this operation takes effect, the node is indeed part of the data structure. Therefore, the node must be persisted *before* the node is inserted into the data structure. Not persisting the node in the correct order would create non-linearizable histories [18].

To accomplish this, PLayer has to keep track of lines that are modified in the CPU's LLC and persist them before the node gets inserted into the queue. Since the data structure is allocated on the FPGA memory space, the FPGA gets notified of each cache line accessed by the CPU during the node creation process. That is, the FPGA would receive an upgrade-to-*Exclusive* request for each cache line that is cached by the CPU in its LLC. This request can be observed by PLayer which then provides access to the data thereby allowing the DC to acknowledge the upgrade request. However, to guarantee that it gets the relevant node's dirty cache lines from the CPU's LLC, immediately after the upgrade request, PLayer issues a *clean* request to the DC to downgrade the line from *Exclusive* to *Shared*-state in the CPU's LLC. This is shown in part 1 of Figure 3 and PLayer does this for every upgrade request from the CPU.

When the CPU receives the upgrade acknowledgement from the FPGA, it modifies the cache line in its cache, and when the CPU receives the downgrade request from the FPGA, it responds with the dirty data. The dirty data is then observed by PLayer and persisted. This way, the FPGA persists all data written by the CPU to the node.

3.3 Insert

As the thread proceeds to insert the node it prepared locally, into the queue, it needs to atomically modify the *next*-pointer of the last node in the queue to point to the node it wants

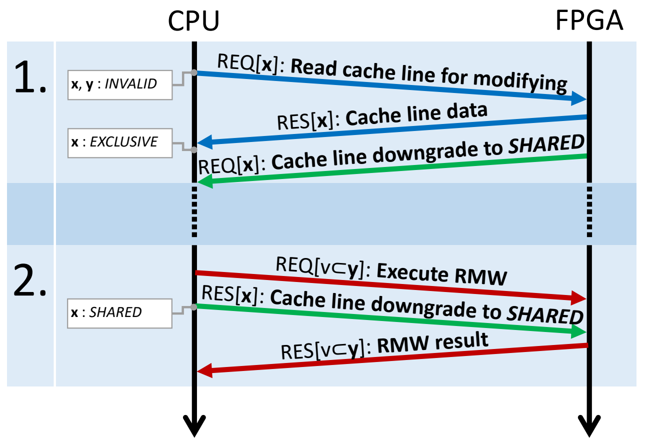
CPU
FPGA
1.
x, y : INVALID
x : EXCLUSIVE
REQ[x]: Read cache line for modifying
RES[x]: Cache line data
REQ[x]: Cache line downgrade to SHARED
2.
x : SHARED
REQ[v⊂y]: Execute RMW
RES[x]: Cache line downgrade to SHARED
RES[v⊂y]: RMW result

Figure 3. PLayer protocol: Sequence of coherence messages exchanged between the CPU and FPGA.

to insert. To achieve this atomically, the CPU issues a read-modify-write (RMW) on the *next*-pointer of the last node in the data structure (Algorithm 1, line 9). Upon executing the RMW, the FPGA gets a special coherence request from the CPU's LLC. PLayer intercepts this request and holds off responding to it until all responses to previously issued *clean* requests are received, as shown in part 2 of Figure 3. This allows PLayer to ensure that all modified data in CPU's LLC are persisted before allowing the CPU to atomically modify the *next*-pointer.

Note that, before the CPU executes the RMW instruction on a particular address, it cannot have the cache-line in *Exclusive* or *Modified* states. In these cases, the FPGA has the cache-line in an *Invalid*-state, and thus the CPU performs the RMW instruction locally in its own cache, without notifying the FPGA. Therefore, the invariant we preserve is that after every RMW instruction, the cache line the CPU operates on remains in an *Invalid* or *Shared* state on the CPU side. When a RMW instruction is executed on the CPU side, assuming the cache line is initially in *Invalid* or *Shared* state, the FPGA gets a message requesting the execution of the RMW operation at the FPGA. Consequently, the FPGA executes the operation in its own memory and sends an acknowledgment for the RMW instruction. Therefore, after a RMW instruction, the cache line remains in its previous state on the CPU.

3.4 Update and delete

Updates or deletes to nodes also follow the same protocol as inserting new nodes; any updates made in the CPU's LLC are tracked and persisted and atomic RMW coherence requests (to modify the *next*-pointer) are delayed until all the modified data using normal writes is persisted.

3.5 Read

The CPU can read data from the data structure through the standard read instruction. If the data is cached in the CPU, it is safe to read since PLayer has already persisted this data before. If the data is not cached in CPU's LLC, PLayer would observe an upgrade request and can provide the most up-to-date value for the data from the persistent memory in a *Shared* state. Having a line in a *Shared* state does not change the PLayer protocol: The cache line is already clean. Finally, since PLayer does not invalidate the cache line, it does not affect the temporal locality for subsequent reads.

3.6 Recovery

Each time the application is launched, it locates the data from the previous execution and traverses the data structure to locate all the reachable nodes. This step is necessary to prevent a memory leak caused by node data, which is already persisted, but not part of the data structure. Furthermore, the application needs to be aware of the existing nodes to avoid running at the risk of overwriting them.

4 Implementation

In the previous sections, we explained the persistence layer protocol. In this section, we will explain the protocol's implementation on the available platform we used, Enzian [11]. This protocol can be adjusted, in the future, to CXL platforms and be implemented inside the CXL's coherence manager.

The FPGA's DC exposes an AXI interface to access the FPGA memory (Figure 2). Any upgrade or atomic RMW request from the CPU can be observed through the AXI's *read-request* channel (with side-band signals distinguishing upgrades and RMW requests). Responses to these requests are issued through the AXI *read-response* channel along with cache line data. Dirty data that is downgraded from the CPU's LLC, on the other hand, can be observed through the AXI's *write-request* channel. Once dirty data is persisted, an acknowledgment can be sent to the DC through the AXI's *write-response* channel.

Moreover, the DC also exposes *local* request and response channels (indicated by LCL in Figure 2) where user logic on the FPGA can issue clean or invalidate requests and receive responses upon the operation's completion. In a system without the persistence layer, the DC's AXI interface is connected directly to the memory controller's AXI interface and the local interfaces are not used.

With the persistence layer, read requests (for upgrading or RMWs data) from the DC are served by the memory controller through the persistence layer. This allows the persistence layer to observe the cache line address being upgraded and issue a clean request to the DC through its *local* interface. Write requests (for downgraded data) from the DC, with dirty data cleaned from CPU's LLC, are also served by the memory controller through the persistence layer. As each clean operation completes, the DC sends an acknowledgment to the persistence layer, allowing it to keep track of the number of outstanding clean operations.

Eventually, when an atomic RMW request arrives on the AXI read-request channel, the persistence layer can delay this request from being issued to the memory controller until there are no outstanding clean operations. This ensures that any dirty data from the CPU's LLC is persisted before, e.g., the node gets added to the lock-free data structure.

The persistence layer has to be designed with enough outstanding DC and memory transactions to saturate both ECI and memory bandwidth. Such a highly concurrent system can suffer from protocol deadlocks if not properly implemented. For example, when the protocol delays a RMW request until all the cleans' acknowledgements arrive, a deadlock can occur if an acknowledgement is stuck behind the stalled RMW request. This is avoided by having independent AXI and local channels that guarantee that an acknowledgement for clean will not arrive in the AXI read-request channel. Other deadlocks due to limited FPGA resources, would have to be considered as part of the implementation.

4.1 Software support

We aim to avoid having to change application code as much as possible, as demonstrated in Algorithm 1. It turns out that, while most of the persistence handling can, in fact, be done in hardware, some software support is still required.

Concretely, we still need software support for allocating the data structure in the FPGA memory space, to get the necessary coherence-messages from the CPU. A dedicated driver exposes the FPGA's memory and allows virtual to physical address translation. Therefore, it should be initially allocated on the FPGA.

Secondly, for being able to recover after a crash, the application needs to know where the data structure resides in the FPGA memory space, and which parts of it hold useful data, to prevent overwriting useful data. This is achieved by always allocating the data structure root in the same place and providing a traversing method, which traverses all the reachable nodes. From the root, the application can traverse the data structure, to find all nodes that are currently part of it. If the allocator supports marking allocated space, the application can add the traversed nodes to the allocator-directory. Otherwise, the nodes found can be reallocated [19]. In any case, after a recovery, only nodes which are part of the data structure will be marked in the allocator. This way, persistent memory leaks are prevented.

4.2 Persistent storage

As we allocate the data structure in the FPGA memory space, we need a way of storing data on the FPGA side, in a persistent manner. Because of the frequent writes and their byte-level granularity, we recommend using Non-Volatile Main Memory (NVMM), such as 3D X-point. This way, we expect to see the best performance. However, there is nothing preventing the use of block devices such as SSDs for persistent storage.

4.3 Proof of concept

For verifying that the proposed protocol works, we conducted several experiments. We used a C program running on a Soft-CPU on the FPGA to decode and respond to cache-coherence messages sent by the CPU. We executed RMW instructions on memory addresses in the FPGA memory space and inferred the relevant ECI message formats and protocol behavior. A similar analysis was done for the downgrade-procedure of cache-lines. The inferred ECI protocol invariants support our PLayer protocol. However, the full aforementioned protocol is still not fully implemented on the Soft-CPU. We still need to combine everything together. Moreover, for further evaluation of the protocol under maximum load, a hardware implementation is required.

5 Optimizations

5.1 DRAM read-cache

In this variant, the FPGA both manages a persistent replica on a persistent storage medium, such as an SSD or NVMM, and a volatile replica in DRAM, similarly to Mirror [19]. This solution benefits from DRAM's low access latency while using a slower device, such as an SSD, for persistent storage.

5.2 Aggregating cache line downgrade requests

Instead of directly following up each response to a cache-line upgrade request by the CPU (e.g., as the result of node preparation) with a downgrade request, we aggregate them until an RMW instruction is signaled to the FPGA. At this point, the FPGA can issue a downgrade request to all cache lines in *Exclusive* or *Modified* state. While this variant requires more bookkeeping on the FPGA side, it prevents unnecessary, repeated up- and downgrading of cache lines. This, in turn, can substantially reduce the number of CPU cache misses. Depending on the frequency of the described issue, it could make sense to choose this variant over handling the upgraded cache lines as described in Section 3.2.

5.3 Exploiting knowledge about the data structure

When a RMW-instruction is executed, the FPGA needs to pull all cache-line data. For some data structures, it might be feasible to exploit data structure knowledge on the FPGA side, and send only the clean requests that are relevant to a particular RMW instruction. In conjunction with Section 5.2, the amount of cache-lines to be downgraded upon receiving a RMW-instruction can be substantially reduced.

6 Related Work

So far, mainly persistence software solutions were proposed. The scope spans from hand-tuned constructions [8, 16, 18, 31] which are the most optimized ones to general constructions [7, 10, 15, 17, 19, 22, 23, 30, 34, 38–40]. General constructions also include transactions and different persistent logging techniques [12, 13]. Ramalhete et al. [33] present two algorithms for persistent concurrent transactions in the form of user space libraries that offer transactional memory semantics. They achieve the lower bound in terms of memory fences needed [14]. By implementing persistent snapshots, this solution provides weaker persistence guarantees than our work, while being applicable to not just lock-free data structures.

Bhardwaj et al. [6] propose using cache-coherent accelerators to implement write-ahead logging in hardware. The authors propose this as a solution for forthcoming commercial cache-coherent accelerators, such as CXL compatible devices. Together with a software library included in the application, the cache-coherent accelerator implements persistent snapshot semantics. Compared to our solution, the proposed hardware accelerated snapshotting does not work in a concurrent setting, without having to serialize all threads for taking a persistent snapshot. It is, however, more general, since it is not just applicable to lock-free data structures.

Ogleari at al. [32] propose a hardware mechanism for undo and redo logging to avoid flushing, by making the logs uncacheable. However, in some cases, a forced write-back mechanism is used for correctness, and software support is required. Moreover, is was only implemented on a simulator.

eADR from Intel is a hardware-mechanism that, in the event of a power-failure, makes sure that all data from the CPU's caches are written to NVMM [1]. This has several advantages compared to traditional software-mechanisms, such as flushing, because the application does not have to wait for these operations to complete, to guarantee correctness. However, with eADR, write buffers are still volatile. Compared to PLayer, eADR still needs explicit fences, e.g., to ensure that local node data hits NVMM before the corresponding node gets inserted. PLayer avoids this problem by pulling the local data from the CPU, before modifying the data structure state.

7 Future Work and Conclusions

We plan to extend Player into a real hardware implementation on Enzian [11]. Then, we will evaluate our hardware-based approach against the state-of-the-art, software transformations and measure the real trade-offs.

With the upcoming, commercially available CXL-enabled CPUs and accelerators, developing a prototype based on CXL will become feasible. In this context, evaluating the requirements of our persistence layer against what the CXL protocol offers will offer interesting insights.

With our proposed hardware-accelerated persistence layer for lock-free data structures, we provide a general solution for making lock-free data structures persistent. The required application-code changes are minimal. Most notably, it does not require any changes to the data structure code itself and lets the hardware handle its persistence.

References

[1] [n. d.]. *eADR: New Opportunities for Persistent Memory Applications.* https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[2] Accessed 2023. Compute Express Link. https://www.computeexpresslink.org/.

[3] Accessed 2023. CrossBar 3D ReRAM. https://www.crossbar-inc.com/products/high-density-memory/.

[4] Accessed 2023. Intel® Optane™ PMem. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html.

[5] Naama Ben-David, Michal Friedman, and Yuanhao Wei. 2022. Survey of Persistent Memory Correctness Conditions. https://doi.org/10.48550/arXiv.2208.11114 arXiv:2208.11114 [cs].

[6] Ankit Bhardwaj, Todd Thornley, Vinita Pawar, Reto Achermann, Gerd Zellweger, and Ryan Stutsman. 2022. Cache-coherent accelerators for persistent memory crash consistency. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems.* ACM, Virtual Event, 37–44. https://doi.org/10.1145/3538643.3539752

[7] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency, Vol. 49. ACM, 433–452.

[8] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. 8, 7 (2015), 786–797.

[9] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. *FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory.* 1077–1091.

[10] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *asplos.*

[11] 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. 2022. 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 '22).* Association for Computing Machinery, New York, NY, USA, 434–451. https://doi.org/10.1145/3503222.3507742

[12] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19).* Association for Computing Machinery, New York, NY, USA, 441–454. https://doi.org/10.1145/3297858.3304046

[13] Nachshon Cohen, Michal Friedman, and James R Larus. 2017. Efficient logging in non-volatile memory by exploiting coherency protocols, Vol. 1. ACM, 67.

[14] Nachshon Cohen, Rachid Guerraoui, and Igor Zablotchi. 2018. The Inherent Cost of Remembering Consistently. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18).* Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3210377.3210400

[15] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. ACM, 271–282.

[16] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures.

[17] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2021. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. arXiv. http://arxiv.org/abs/2004.02841 arXiv:2004.02841 [cs].

[18] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, Vienna Austria, 28–40. https://doi.org/10.1145/3178487.3178490

[19] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. Mirror: making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* ACM, Virtual Canada, 1218–1232. https://doi.org/10.1145/3453483.3454105

[20] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[21] PMDK Intel. 2018. *Persistent Memory Programming.* https://pmem.io

[22] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16).* Association for Computing Machinery, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[23] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing*, Cyril Gavoille and David Ilcinkas (Eds.). Vol. 9888. Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327. https://doi.org/10.1007/978-3-662-53426-7_23 Series Title: Lecture Notes in Computer Science.

[24] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. 2016. High-performance transactions for persistent memories. 399–411.

[25] R Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. {TIPS}: Making Volatile Index Structures Persistent with {DRAM-NVMM} Tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21).* 773–787.

[26] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. 257–270.

[27] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building durable transactions with decoupling for persistent memory. ACM, 329–343.

[28] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[29] Virendra Marathe, Achin Mishra, Amee Trivedi, Yihe Huang, Faisal Zaghloul, Sanidhya Kashyap, Margo Seltzer, Tim Harris, Steve Byan, Bill Bridge, et al. 2018. Persistent memory transactions. *arXiv preprint arXiv:1804.00701* (2018).

[30] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. 789–806.

[31] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 91)*, Andréa W. Richa (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 37:1–37:16. https://doi.org/10.4230/LIPIcs.DISC.2017.37

[32] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 336–349. https://doi.org/10.1109/HPCA.2018.00037

[33] Pedro Ramalhete, Andreia Correia, and Pascal Felber. 2021. Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of*

Parallel Programming (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3437801.3441586

[34] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory.

[35] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: an easy-to-use Java NVM framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 316–332.

[36] Viking Technology. 2017. *Persistent Memory Technologies*. https://www.vikingtechnology.com/products/nvdimm/

[37] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory., Vol. 11. 61–75.

[38] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory, Vol. 39. ACM, 91–104.

[39] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2021. FliT: A Library for Simple and Efficient Persistent Algorithms. http://arxiv.org/abs/2108.04202 arXiv:2108.04202 [cs].

[40] Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. 2021. A Fast, General System for Buffered Persistent Data Structures. In *Proceedings of the 50th International Conference on Parallel Processing (ICPP '21)*. Association for Computing Machinery, New York, NY, USA, Article 73, 11 pages. https://doi.org/10.1145/3472456.3472458

[41] Xilinx. 2022. Xilinx Zynq Ultrascale+ MPSoC. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[42] Jian Xu and Steven Swanson. 2016. {NOVA}: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. 323–338.

[43] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-Execute More. 346–359.

[44] Pantea Zardoshti, Tingzhe Zhou, Yujie Liu, and Michael Spear. 2019. Optimizing Persistent Memory Transactions. IEEE, 219–231.

[45] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets.