CPU-Oblivious Offloading of Failure-Atomic Transactions for Disaggregated Memory

Cheng Chen*
Huazhong University of Science and Technology
Wuhan, Hubei, China
chencheng@hust.edu.cn

Chencheng Ye*
Huazhong University of Science and Technology
Wuhan, Hubei, China
yecc@hust.edu.cn

Yuanchao Xu
University of California, Santa Cruz
Santa Cruz, California, USA
yxu314@ucsc.edu

Xipeng Shen
North Carolina State University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

Xiaofei Liao*
Huazhong University of Science and Technology
Wuhan, Hubei, China
xfliao@hust.edu.cn

Hai Jin*
Huazhong University of Science and Technology
Wuhan, Hubei, China
hjin@hust.edu.cn

Wenbin Jiang*
Huazhong University of Science and Technology
Wuhan, Hubei, China
wenbinjiang@hust.edu.cn

Yan Solihin
University of Central Florida
Orlando, Florida, USA
Yan.Solihin@ucf.edu

Abstract

Memory disaggregation introduces new challenges for application reliability, as compute server or interconnection failures can interrupt execution and lead to data inconsistency in the memory server. This paper presents Fanmem, a novel failure-atomic transaction system designed specifically for disaggregated memory architectures. Fanmem ensures data consistency in the presence of failures, drawing inspiration from persistent memory transactions while tailored for memory disaggregation. The key innovations of Fanmem include an asynchronous transaction model and the integration of a processing unit within the switch, enabling the offloading of time-consuming log persistency operations to the switch processing unit and significantly reducing the overhead on the compute servers. Evaluation confirms the effectiveness of Fanmem on two representative memory-disaggregated architectures. Compared to the state-of-the-art persistent memory transaction system, Fanmem achieves an average performance improvement of 1.2× and 1.7× on the respective architectures.

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **Software and its engineering → Software reliability**.

Keywords: Persistent Memory, Disaggregated Memory, Crash Consistency

ACM Reference Format:
Cheng Chen, Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Yan Solihin. 2026. CPU-Oblivious Offloading of Failure-Atomic Transactions for Disaggregated Memory. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790146

1 Introduction

The landscape of datacenter architecture is undergoing a fundamental transformation, driven by the disaggregation of compute and memory resources. Enabled by emerging high-bandwidth and low-latency interconnects such as *Compute Express Link* (CXL), this new paradigm allows multiple compute servers to access a large memory pool [10, 20, 35, 41]. This approach promises substantial improvements in memory utilization and cost-efficiency, mitigating the chronic problem of memory stranding and allowing systems to be provisioned with performant, large-capacity, and cost-effective remote memory. Concurrently, the *Persistent Memory* (PM) landscape is evolving; following the discontinuation of Intel's Optane PMEM, the industry is shifting

*Chencheng Ye is the corresponding author. Cheng Chen, Chencheng Ye, Xiaofei Liao, Hai Jin, and Wenbin Jiang are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790146

toward CXL-attached solutions like memory-semantic SSDs such as Samsung CMM-H [53, 68] and UPS-backed DRAM servers [18], which integrate seamlessly into these disaggregated architectures.

Persistent memory transactions are a fundamental abstraction for managing the crash consistency of PM, but existing designs are ill-suited for the unique characteristics of CXL-based disaggregated memory. The core research problem lies in the prohibitive latency of persistence operations. Accessing CMM-H remote memory via CXL can take thousands of CPU clock cycles, e.g., 728.9ns median latency [68]. Such high latencies greatly magnify the cost of persist barriers (e.g., `sfence`), which stall the CPU pipeline and severely degrade application performance [35], as shown in Figure 1 (a).

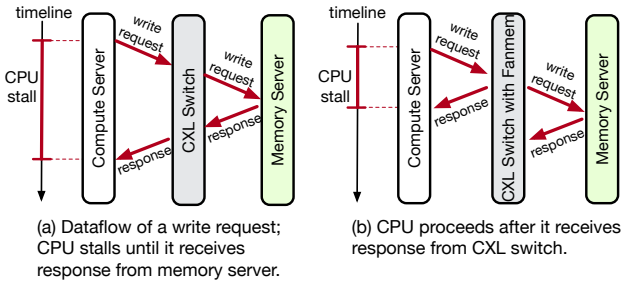

Figure 1. Dataflow of (a) a standard synchronous write request and (b) an offloaded write request

Current persistent memory transaction systems fail to adequately address this challenge. Software-only approaches, which focus on minimizing persistence operations [66] or moving them to a background thread [39], remain bottlenecked by the high-latency synchronous flushes required to ensure durability over the CXL fabric, resulting in significant overhead (an average slowdown of 1.6×) compared to persistent memory transactions running on local memory, as shown in Figure 2. Hardware-assisted solutions, on the other hand, often achieve higher performance by integrating transaction logic into the CPU [6, 13]. However, these designs require non-trivial modifications to the processor core, limiting their adoption in heterogeneous datacenters deploying a wide range of CPUs, including custom silicon from major cloud providers [4, 8, 36, 38]. A CPU-centric solution is fundamentally at odds with the vendor-agnostic philosophy of CXL, making a CPU-oblivious approach highly desirable.

In this paper, we propose Fanmem, the first failure-atomic transaction system for CXL-based disaggregated memory architectures with hardware-accelerated crash consistency. The core idea of Fanmem is a software-hardware co-design that decouples the transaction's execution on the compute server from its log persistence. We introduce an asynchronous transaction model that offloads the time-consuming log persistence operations to a Fanmem-enabled CXL switch, as shown in Figure 1 (b). This offloading moves the latency of

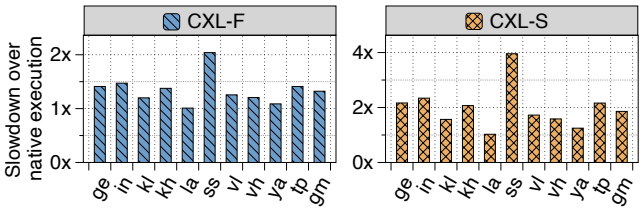

Figure 2. Slowdown of software speculatively persistent memory transaction [66] on two disaggregated memory architectures (CXL-F [35] and CXL-S [12]) over the native execution on local memory. CXL-F and CXL-S differ in the latencies between the CXL switch and the memory server. Detailed parameters are in Section 6.1. *gm* denotes geomean.

writing to the remote memory server off the critical path, allowing the application to proceed without stalling. Transaction commits are delayed, enabling unrelated subsequent tasks to execute before commit completion while preserving durability semantics. The novelty of Fanmem lies in its CPU-oblivious design; by embedding the acceleration logic into the network fabric (the CXL switch), it remains compatible with any CXL-compliant CPU, ensuring broad applicability.

This offloading strategy presents several challenges. First, the CXL switch must efficiently distinguish offloaded log writes from the high volume of normal memory traffic. Fanmem addresses this challenge with a hardware *Log Area Table* (LAT) that performs fast lookups on the address ranges of registered log areas. Second, the application must be able to check for persistence completion without incurring the high overhead of system calls or interrupts. We solve this problem with an append-only sequential log and a hardware-managed *cursor* in the switch. This cursor tracks the durable log frontier and is exposed to the application via a simple memory-mapped load instruction, enabling an efficient, low-overhead tracking mechanism. Third, to ensure correctness when logs persist out of order, Fanmem employs a timestamp-based commit protocol, guaranteeing that transactions are committed in the same order as they were executed.

Our evaluation on an FPGA-based prototype and a gem5-based simulation of two representative disaggregated memory architectures demonstrates the effectiveness of Fanmem at reasonable resource overhead. Using applications from the STAMP and TPC-C benchmarks, Fanmem achieves average performance improvements of 1.2× and 1.7×, respectively, compared to a state-of-the-art software persistent memory transaction system [66].

2 Background

This section provides the necessary background on CXL-based memory disaggregation and persistent memory transactions.

2.1 CXL-Based Memory Disaggregation

CXL is an emerging industry standard that enables high-bandwidth and low-latency interconnectivity among CPUs, NICs, accelerators, and other devices. It introduces a new attach point for memory, allowing CPUs to access memory attached to other devices through load and store instructions. Such capability facilitates memory disaggregation, such that multiple compute servers with powerful CPUs can connect to a memory server with large memory capacity through a CXL switch, as shown in Figure 3.

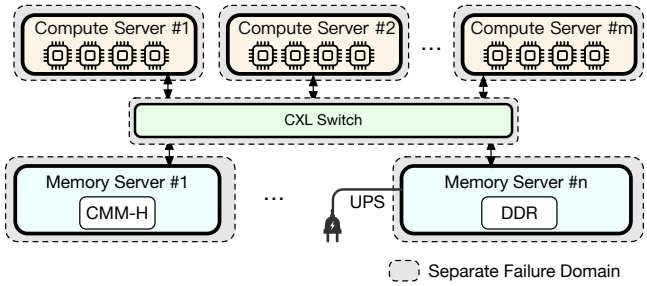

Figure 3. An overview of a memory disaggregated architecture. Multiple compute servers share memory servers via a CXL switch. The CPU-free memory server contains various types of memory. The compute servers, memory servers, and switch reside in separate failure domains.

The compute server, memory server, and CXL switch are independent devices [20] that reside in separate failure domains. However, such memory disaggregation introduces potential reliability challenges, as the failure of any component, whether it be the compute server, the CXL switch, or the memory server, can lead to interruptions in memory-disaggregated applications. As a reference, a study conducted by Microsoft on 180,000 switches across 130 locations reported a failure rate of 2.5% for data center Ethernet switches within an 80-day period [52]. This statistic underscores the real-world impact of component failures on the reliability of systems utilizing disaggregated memory.

2.2 Persistent Memory Transaction

Persistent memory transactions are a widely adopted abstraction for ensuring failure atomicity. A transaction groups a sequence of memory operations, guaranteeing that either all modifications are durably persisted or none are, even in the event of a crash. This all-or-nothing semantic is typically implemented using a logging protocol. For example, consider a transaction that updates *a*. Undo logging records the old value of *a* before modifying it. The transaction persists the data using a cache flush `clwb` and a write barrier `sfence` before committing, as shown in Figure 4 (a). Upon system failure, post-failure recovery utilizes the undo log records to revert the modifications if the transaction is uncommitted.

Persistent memory transactions can be implemented with different logging schemes that vary in performance and memory space consumption, including undo [16, 28, 30, 51, 55, 66] and redo [13, 14, 26, 39]. Although effective, these traditional transaction models incur significant performance overhead in a disaggregated memory setting, as shown in Figure 2. The core bottleneck is the high latency of persistence [48].

To mitigate this overhead, recent research has proposed *speculative logging* [66]. As illustrated in Figure 4 (b), speculative logging restructures operations to reduce the number of expensive fences. By logging the new value of data `a` in Transaction #1 and using its fence to ensure persistence, a subsequent Transaction #2 that also modifies `a` can leverage this persisted log for its own recovery, eliminating the need for its own log persistence fence. Since the log already contains the new data, the data write itself does not need to be persisted before commit.

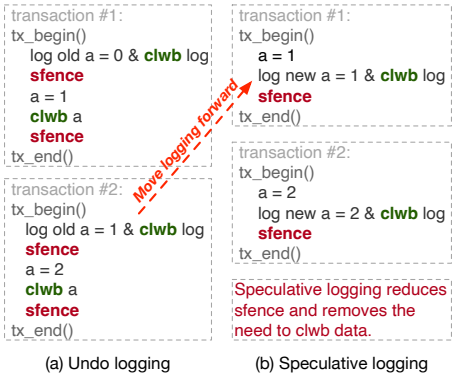

Figure 4. Comparison of (a) undo logging and (b) speculative logging

This optimization significantly reduces overhead by requiring only one `sfence` per transaction. However, a fundamental performance challenge remains: this single fence is still synchronous and stalls the CPU until the log data traverses the high-latency network and is persisted to the remote memory server. It is this critical-path latency that Fanmem is designed to eliminate.

3 Overall Design

The fundamental performance bottleneck for transactions in a disaggregated memory environment is the high latency of synchronous persistence operations. The core design principle of Fanmem is to decouple transaction execution from log persistence, thereby moving the high cost of remote durability off the application's critical path. We achieve this through a software-hardware co-design that introduces an asynchronous transaction model in software and offloads the management of log persistence to a lightweight, intelligent hardware unit embedded within the CXL switch. Placing the acceleration logic in the switch is a strategic choice; as a

natural interception point for all memory traffic, it allows Fanmem to remain *CPU-oblivious*. This stands in contrast to previous hardware-assisted solutions that required custom CPU modifications, limiting their practical adoption [6, 13].

3.1 Asynchronous Transaction Model

Fanmem implements an asynchronous transaction model in which persistence is determined through explicit, asynchronous verification of persistence progress. Under this model, data achieves persistence only upon reaching persistent memory, as both compute servers and switches are treated as volatile components. Transactions can commit only after their persistence has completed. Before commitment, our solution allows software and hardware to persist log records in arbitrary order, providing flexibility in the logging mechanism while maintaining correctness guarantees.

Similar to other asynchronous transaction schemes [6, 26, 30, 39, 42], Fanmem's software interface is built upon an asynchronous transaction model that divides a transaction's lifecycle into three distinct stages: execution, offloading, and commit. To illustrate this, we integrate Fanmem with a state-of-the-art speculative logging scheme [66], though the principle is applicable to other schemes as discussed in Section 5.1. Figure 5 shows the modified programming model.

```
int txid = tx_begin()  // assigns a unique ID
  a = 1
  log(&a, 1)   // create a log record          1.Execution
  clwb log     // offload to CXL switch
  sfence                                       2. Offloading
tx_end()
... // application proceeds with other tasks
if(isPersisted(txid)):  // check persistency
  tx_commit(txid)       // commit transaction   3. Commit
```

Figure 5. The asynchronous transaction model in Fanmem, which divides a transaction's lifecycle into execution, offloading, and commit stages. By offloading log persistence, the application avoids stalling on remote memory durability.

1. Execution: The application begins a transaction with `tx_begin()` and performs its data modifications directly. In line with speculative logging, it generates log records containing the new data values.

2. Offloading: The application writes the log records to a pre-allocated region in disaggregated memory. Crucially, it then executes a `clwb` followed by an `sfence`. In a traditional system, this `sfence` would stall the CPU until the log data is durable in the remote memory server. With Fanmem, the semantics are transformed. The fence only needs to ensure that the log write requests have reached the *Fanmem-enabled CXL switch*. Upon receiving these writes, the switch immediately buffers them and sends an acknowledgment back to the compute server. This early acknowledgment allows the CPU to retire the fence instruction and proceed with subsequent work—including executing new transactions—long before the logs are physically persisted. This stage concludes with `tx_end()`, signifying that the transaction's execution is complete and its persistence has been successfully offloaded.

3. Commit: The application cannot finalize the transaction until it can confirm that its logs are truly durable. Fanmem provides a low-overhead polling mechanism for this purpose, i.e., `isPersisted(txid)`. Once persistence is confirmed, the application calls `tx_commit(txid)` to formally commit the transaction, making its effects permanent and visible for recovery.

The central challenge of this asynchronous model is enabling the application to perform the `isPersisted()` check efficiently and accurately, without introducing new performance bottlenecks.

3.2 The Sequential Log and Cursor Mechanism

To solve this challenge, Fanmem introduces a simple yet powerful logging discipline: a per-thread, append-only sequential log area. The application allocates a dedicated, physically contiguous region of disaggregated memory for each application thread. The thread then writes all its log records sequentially into this area.

This structural constraint is the key insight that simplifies persistence tracking. Instead of managing the status of numerous individual and scattered log writes, the problem is reduced to tracking a single "high-water mark" of durability for each log area. Fanmem accomplishes this with a hardware-managed cursor.

The cursor is a pointer maintained in the Fanmem hardware within the CXL switch, with one cursor dedicated to each offloaded log area. This cursor's value represents a critical guarantee: *all log data from the beginning of the log area up to the address held by the cursor has been confirmed as durable on the remote memory server.*

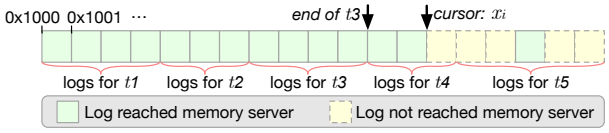

Figure 6. A sequential log area starting from 0x1000. The logs for transaction *t3* and preceding transactions have reached memory server.

The process for an application to check a transaction's persistence is therefore highly efficient, as shown in Figure 6:

1. During the execution stage, the application software notes the address of the *last byte* of the final log record for the transaction.
2. To check for persistence, the application issues a single `load` instruction to a specific, memory-mapped address that corresponds to its log area's cursor. This request is routed to the switch, which returns the current cursor value.

3. The application then performs a simple comparison: persistence is guaranteed if the transaction's last log address is less than or equal to the returned cursor.

If this comparison is true, the transaction is guaranteed to be persistent. Because the log is sequential and append-only, the durability of its last byte implies the durability of all its preceding bytes. This mechanism transforms a complex synchronization task into a single, low-latency memory read and a comparison, completely avoiding high-overhead system calls or interrupts.

3.3 Hardware-Assisted Offloading

Fanmem augments the CXL switch with two specialized, lightweight hardware components, as shown in Figure 7, the *Log Area Table* (LAT) and the *Log Write Status Table* (LWST), which work in concert to manage the offloaded persistence operations transparently.

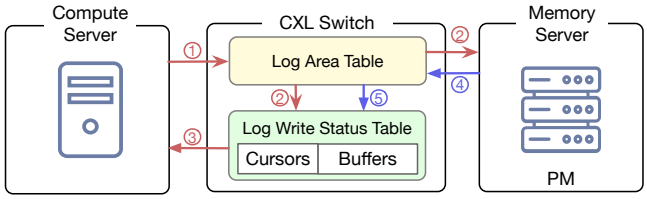

Figure 7. The overall architecture of Fanmem. Log writes are intercepted by the Fanmem-enabled CXL switch, which provides an early acknowledgment to the compute server before forwarding the write to the memory server.

Fanmem serves write requests in the following way:
① When a memory write arrives from a compute server, the switch first consults the *Log Area Table* (LAT). The LAT is a small on-chip cache that stores the physical address ranges of all registered log areas. If the write address is not found in the LAT, it is treated as a normal data write and is forwarded transparently to the memory server. ② Otherwise, the switch identifies it as a log write. Fanmem immediately forwards the write to the memory server for persistence while simultaneously recording the request in its internal *Log Write Status Table* (LWST). ③ Crucially, the switch then sends an immediate completion acknowledgment back to the compute server. This is the mechanism that enables the decoupling described in the transaction model.

The LWST tracks the state of all in-flight log writes. ④ When a write completes in the memory server and the response arrives from it, the switch intercepts this response, ⑤ updates the corresponding buffer in the LWST, and advances the log area's cursor to the new contiguous boundary of durable writes. It is this final step that makes the persistence guarantee visible to the application through the cursor mechanism.

3.4 Commit Protocol

To ensure correctness, a transaction can only commit after two conditions are met: (1) all of its log records have been durably persisted, and (2) all transactions that it depends on have also been committed. Fanmem defines dependencies conservatively based on execution order: a transaction depends on all transactions that completed their execution stage earlier.

To enforce this, Fanmem employs a timestamp-based commit protocol, similar in principle to prior works [14, 66]. When a transaction finishes its execution stage, the application records a hardware timestamp. To commit, a thread checks that its transaction's logs are persistent (via the cursor) and then coordinates with other threads to find the oldest uncommitted transaction that is ready. The system-wide commit timestamp is then advanced, atomically committing all transactions with earlier timestamps. This protocol guarantees that transactions are committed in the same order they were executed, ensuring that post-failure recovery will restore a consistent system state. All timestamp management and commit logic resides on the CPU; the switch remains oblivious to transaction semantics, preserving the simplicity of the hardware design.

When the system scales out with more switches, Fanmem works in the same way, as it treats each process independently. A compute server offloads transactions to the nearest switch equipped with Fanmem, regardless of the topology of the switches.

4 Implementation

This section details the hardware and software co-design of Fanmem.

4.1 Log Area Table

A CXL switch may connect numerous compute servers, each generating a high volume of memory traffic [2, 12]. The hardware must efficiently distinguish latency-sensitive log writes from regular data writes with minimal overhead and without becoming a bottleneck, while avoiding any modification to the CXL protocol. A monolithic, fully-associative lookup structure would be prohibitively slow and complex.

To tackle the challenge of scalable traffic interception, Fanmem employs a partitioned and dynamically managed LAT structure instead of a single, large lookup table. The LAT subsystem consists of two main components, as shown in Figure 8: an *Index Table* and multiple smaller, independent *pLATs*. The Index Table is a simple SRAM array indexed by the source port ID (i.e., the ID for the compute server initiating the write request, defined by the CXL protocol) of an incoming memory write request. Each entry in the Index Table is an index of a pLAT. Each pLAT is a range-based TLB [29] that stores the base physical address and size (in 4KB granularity) of registered log areas.

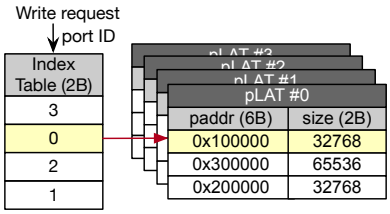

Figure 8. Design of the Log Area Table for efficient identification of log writes

When a write request arrives the switch, its source port ID is used to perform a fast, direct lookup in the Index Table to identify the pLAT. A parallel lookup is then performed within the selected pLAT to check if the write's address falls within a registered log range. If there is a hit in the pLAT, the write is identified as a log write and is forwarded to the Log Write Status Table for asynchronous handling. If there is a miss, the write is treated as normal traffic and is passed transparently to the memory server.

Both the Index Table and pLAT are memory-mapped. The assignment of pLATs is managed by a management node (master). Specifically, the master configures the Index Table associated with the compute server to reflect the ID of the corresponding pLAT. This method offers two main advantages: First, it allows multiple compute servers, each offloading only a few log areas, to share the same pLAT, thus reducing underutilization of pLAT entries. Second, the master has the flexibility to dynamically remap a compute server to another pLAT. Thus, when the number of offloaded log areas associated with a compute server fluctuates, the master can reassign it to a shared or dedicated pLAT, preventing pLAT underutilization or overflow. If the pLAT is exhausted, Fanmem simply resorts to traditional persistency, i.e., it signals log write completion after it reaches the memory server.

The application manages the update of pLAT entries allocated to it. Upon registering offloaded log areas, each thread writes the starting address and the size of the designated log area into the corresponding pLAT entry. During log area reclamation, the corresponding pLAT entry is removed, reflecting the deallocation of the associated memory region.

4.2 Log Write Status Table

The LWST's architecture mirrors the partitioned design of the LAT. It is composed of multiple independent tables, each called a pLWST, with each entry dedicated to a single log area. A log write forwarded from the LAT is routed to its corresponding pLWST using metadata from the LAT lookup (the pLAT's index and row ID), eliminating the need for further associative searches.

Each pLWST entry leverages the sequential, append-only nature of the logs to implement a highly compact, stateful tracking mechanism. Instead of storing full addresses for every pending write, it maintains two key components, as shown in Figure 9:

- A **cursor**, which stores the base address of the oldest in-flight log write.
- A **sliding window**, implemented as a bit-vector that tracks the status of a contiguous block of subsequent 64-byte log entries starting from the position referred by the cursor. We use a 2-bit, three-state encoding for each entry: *empty* (not yet received), *buffered* (received by the switch but not yet durable in memory), and *persisted* (confirmed durable by the memory server).

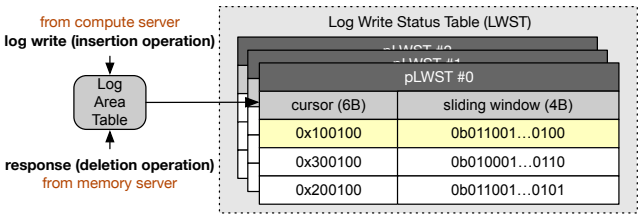

Figure 9. Architecture of the Log Write Status Table, which tracks in-flight log writes

The functionality of the LWST comprises three operations:

Insertion: When a log write arrives, the LWST calculates its offset from the cursor associated with the same log area, updates the corresponding bit-vector entry to "buffered", and immediately sends an acknowledgment to the compute server, indicating that the write has reached the switch. Later, the switch forwards the write to the memory server.

Persistence Confirmation: When the memory server confirms a write is durable, the LWST marks the corresponding entry as "persisted". It then inspects the head of the sliding window. The cursor is advanced across all contiguous "persisted" entries, updating the durable log frontier.

Cursor Query: The application polls the durable frontier by issuing a memory-mapped load to an address corresponding to the cursor. The LWST returns the current cursor value, which the software uses to confirm transaction persistence.

The design of the LWST is extremely lightweight. The stateful sliding-window mechanism drastically reduces the on-chip storage required compared to naively buffering full addresses, making it suitable for the resource-constrained switch environment. Direct-mapped access ensures low-latency, contention-free operations.

4.3 Commit Protocol

While the hardware handles persistence, the software is responsible for ensuring correctness. Fanmem uses a timestamp-based commit protocol to guarantee that transactions are committed in the same order they were executed, which is critical for predictable post-failure recovery.

The commit protocol operates as follows:

1. **Timestamping:** When a transaction completes its execution stage (before commit), the application assigns it a per-process timestamp, which can be implemented with the `rdtscp` instruction [14, 66] on x86 or a monotonically increasing counter as a logical timestamp. Because Fanmem operates without cross-process/server timestamp synchronization requirements, it eliminates the need for expensive, high-precision global timers that would otherwise impose significant overhead in disaggregated memory systems.
2. **Dependency Tracking:** Dependencies are defined conservatively by execution order: a transaction depends on all transactions with an earlier timestamp.
3. **Committing:** Each application thread maintains a sorted list of its own completed but uncommitted transactions. To commit, the application finds the transaction with the globally oldest timestamp across all threads that is ready (i.e., its logs have been confirmed persistent via the cursor mechanism). The system then advances a single, persisted, per-process commit timestamp to atomically commit that transaction and all other ready transactions with older timestamps.

Fanmem uses this protocol to enforce ordering while keeping all transaction-level semantics on the CPU. The switch remains oblivious to transaction dependencies, preserving the simplicity and generality of the hardware design. The protocol also naturally allows for batching, where multiple contiguous, ready transactions can be committed together by advancing the timestamp, further improving efficiency.

Since Fanmem commits transactions by updating the per-process timestamp, which can be set to any point t without compromising correctness, provided the application guarantees that all transactions completing execution before t are ready to commit, that is, the transactions are persisted and acknowledged by the application. For example, consider two threads, each containing two executed but uncommitted transactions, with timestamps of 2 and 5 for the first thread, and 1 and 3 for the second, as shown in Figure 10. If all four transactions have been persisted and the application updates the per-process timestamp from 0 to 5, the transactions will commit together. However, if the second thread is still persisting a third transaction with a timestamp of 4, the application should not update the per-process timestamp to 5 because a crash would render the third transaction irrecoverable, and the dependent transaction has been committed.

4.4 Post-Failure Recovery

Fanmem's design provides a unified recovery mechanism for failures affecting the compute server, the CXL switch, or the memory server. The transaction log itself resides in persistent memory within a memory server. In the event of any component failure, the system initiates recovery by reading the last persisted per-process commit timestamp. The commit protocol guarantees that all log entries associated with transactions that have older timestamps are durable in the memory server. The recovery process simply replays these committed logs to restore the system to a consistent state, effectively ignoring any partially completed operations or in-flight data lost in the volatile buffers of a failed switch.

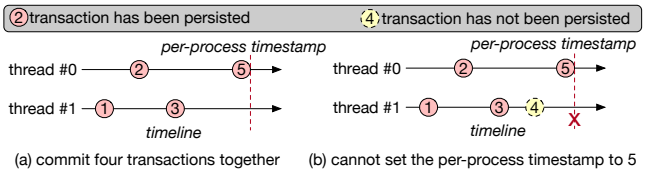

Figure 10. Two cases of updating the per-process timestamp where all transactions are executed but not committed. The number in each circle represents the timestamp obtained at `tx_end()`.

4.5 Performance Guarantees

A key challenge in any buffered, asynchronous system is handling high-contention scenarios where hardware resources become exhausted. In Fanmem, a burst of logging activity from many memory servers could potentially saturate the LAT, causing subsequent log-offload requests to fail.

To ensure robust behavior, Fanmem incorporates a dynamic fallback mechanism that establishes a performance lower bound. When an application thread attempts to offload a log write but fails to acquire an entry in the LAT (i.e., the table is full), it does not stall. Instead, the software runtime seamlessly reverts to a conventional, synchronous persistence protocol for that specific transaction. The log write is still sent, but the CPU will wait for the durability confirmation from the remote memory server, just as it would in a baseline system without Fanmem's offloading capabilities. This strategy ensures forward progress and guarantees that Fanmem's performance will, in the worst case, be no worse than that of the underlying synchronous transaction scheme.

5 Discussion

The core principle of Fanmem, decoupling transaction execution from log persistence by offloading durability management, is a versatile concept not limited to the specific implementation detailed in this paper. The strategic decision to place lightweight acceleration logic in the CXL switch offers significant flexibility, allowing Fanmem's approach to be adapted to various logging protocols, hardware integration points, and disaggregated memory configurations. This section explores the broader applicability and design trade-offs of our approach.

5.1 Applicability to Diverse Logging Protocols

While this paper demonstrates Fanmem's effectiveness in accelerating a state-of-the-art speculative logging scheme [66],

its asynchronous offload model can be extended to other popular logging protocols.

Redo Logging. Fanmem is naturally suited to accelerate redo logging schemes. Many redo-based systems already persist logs asynchronously to mitigate performance overhead [39, 70]. Integrating Fanmem is straightforward, requiring only minor software modifications to interface with the asynchronous offload and commit mechanism. As shown in our evaluation (Section 6), applying Fanmem to a redo logging scheme like SPHT [14] yields substantial performance gains, underscoring its broad compatibility.

Undo Logging. Applying Fanmem to undo logging presents a challenge. Undo logging mandates a strict ordering between a data write and its corresponding log write to ensure atomicity. Offloading these operations would require the hardware to understand and enforce these write-after-write dependencies. While this is achievable, it would require more complex dependency tracking logic within the switch, similar to mechanisms found in CPU-centric hardware transaction systems [6, 17]. Such a design would compromise Fanmem's core philosophy of maintaining a simple, lightweight, and stateless hardware accelerator, which is critical for a latency-sensitive network component.

5.2 Flexibility in Hardware Integration

The placement of acceleration logic is a critical design decision. We contend that the CXL switch is a good location for Fanmem, but other integration points are possible, each with distinct trade-offs.

In the CXL Switch (Our Approach). Placing Fanmem in the switch strikes an ideal balance. As a natural interception point for all memory traffic, it allows Fanmem to be entirely CPU-oblivious, ensuring compatibility with any CXL-compliant processor.

In the Memory Server Controller. Integrating Fanmem logic into the memory controller on the memory server is another possibility. However, this placement would incur the full round-trip network latency for the offload acknowledgment to travel from the compute server to the memory server and back. Our analysis shows this high latency negates most of the benefits, yielding only a marginal performance improvement of 1.2%.

In the CPU. A CPU-centric implementation is the traditional approach for hardware-assisted persistence [6, 13]. While this could offer the lowest latency for offloading, it comes at the cost of requiring non-trivial, bespoke modifications to the processor core.

5.3 Adaptability to System Configurations

An alternative configuration is for a compute server to use its own local persistent memory. This setup is viable for applications with small data footprints that can recover quickly from a local checkpoint. However, this approach forgoes the primary benefits of disaggregation, namely superior memory utilization, on-demand provisioning, and the ability to share large memory resources across many compute servers [35, 46]. For the large-scale, memory-intensive applications prevalent in modern datacenters, a disaggregated model is increasingly favored, making a solution like Fanmem a critical enabling technology for reliable and high-performance computing.

6 Evaluation

We comprehensively assess Fanmem's performance, scalability, and robustness, demonstrating that by offloading log persistence to the CXL switch, Fanmem can significantly mitigate the high-latency overhead of failure-atomic transactions in disaggregated memory environments.

6.1 Methodology

6.1.1 Experimental Platforms. We conduct our primary performance evaluation using a detailed system simulator, Gem5 (v21.2) [11, 32, 56]. To capture the diversity of potential deployments, we evaluate two representative disaggregated memory architectures derived from recent academic and industry proposals:

- **CXL-F:** A fast, switch-based topology modeling a lightweight disaggregated architecture [35]. It features lower latencies, representing systems where compute and memory servers are in close proximity.
- **CXL-S:** A rack-scale architecture with higher network latency, based on performance studies from Micron and AMD [12]. This configuration represents a larger, more scaled-out deployment.

To validate the implementation feasibility and quantify the hardware overhead of our proposed switch modifications, we develope a hardware prototype of Fanmem on a Xilinx U250 FPGA [5]. The design is synthesized using the Xilinx Vitis HLS flow and meets a target frequency of 413 MHz. This prototype provides the basis for the hardware resource consumption analysis presented in Section 6.9 and confirms that our design is practical and lightweight.

6.1.2 System Configuration. Table 1 summarizes the key parameters of our simulated system, configured to reflect a modern datacenter server. The compute server is equipped with out-of-order CPUs and a standard cache hierarchy. We model three types of memory technologies for the memory server to analyze Fanmem's performance across different hardware generations: high-performance UPS-backed DRAM, legacy Persistent Memory [66], and a CXL-attached memory-semantic SSD (CMM-H) [68]. The specific latencies for the CXL fabric and memory servers are detailed in the table and are derived from published measurements and projections [35, 37, 54, 66, 68]. The latencies for Fanmem's components are confirmed by the FPGA-based prototype. The pLAT and pLWST are configured with 32 entries. We

recommend sizing these structures to match the number of cores available on the compute server.

Table 1. System configuration parameters

Component	Parameter
CPU	Out-of-order X86 cores@4GHz, 36 cores
L1 Cache	Private 48KB L1D, 32KB L1I per core
L2 Cache	Private 2MB per core
L3 Cache	Shared 60MB
Local DRAM	DDR4@2400MHz; 14-14-14-32-15 timings
CXL Fabric	PCIe 6.0, 16x lanes (128GB/s) Compute server to CXL switch: 80ns *CXL-F:* Switch to memory server: 115ns *CXL-S:* Switch to memory server: 445ns
Memory Server	UPS-backed DRAM: Same as local DRAM PM: 150ns/500ns read/write latency [66] CMM-H: 728.9ns access latency [68]
Fanmem	LAT: 32,768 entries, 6 cycles lookup LWST: 32,768 entries, 2-3 cycles update

6.1.3 Workloads. In line with recent studies [14, 15, 19, 57, 66], we select a diverse set of workloads to comprehensively evaluate Fanmem across various transactional patterns from the STAMP benchmark [44]. We also evaluate a broadly used [14, 39, 59, 62, 70] online transaction processing application, TPC-C [1]. Consistent with previous work [14], we configure 95% of payment, 2% of new order, and 3% of delivery transactions. Table 2 lists the applications. By default, a single-threaded configuration is used, and Section 6.3 reports multi-threading results. The applications' heap data is placed on the disaggregated memory using libvmem [3], while other data, such as stack variables, remain in the local memory on the compute server.

Table 2. Applications used in evaluation

Applications	Description
genome (ge)	Construct gene sequence
intruder (in)	Detect network intrusion
kmeans-low (kl)	K-means clustering
kmeans-high (kh)	K-means with higher contention
labyrinth (la)	Routing in a maze
ssca2 (ss)	Large directed graph representation
vacation-low (vl)	Transaction processing system
vacation-high (vh)	High contention transaction processing
yada (ya)	Delaunay mesh refinement algorithm
TPC-C (tp)	Online transaction processing

6.1.4 Evaluated Schemes. We compare Fanmem against state-of-the-art schemes to provide a robust evaluation:

- SpecPMT [66]: The default baseline. A state-of-the-art speculative logging scheme.
- Crafty [19]: A state-of-the-art undo logging scheme.
- SPHT [14]: A state-of-the-art redo logging scheme that blocks the software until transaction commits.
- DudeTM [39]: A state-of-the-art asynchronous redo logging scheme that persists data and logs asynchronously.
- PMNet [48]: Extending the persistence domain to network by attaching persistent memory to the NIC.
- No-log: Transaction without logging and persistence.
- Fanmem: The proposed system that accelerates SpecPMT.

6.2 Overall Performance

Fanmem outperforms the baseline on both CXL-F and CXL-S architectures, achieving an average throughput of 1.2× and 1.7×, respectively, as illustrated in Figure 11. The significantly higher throughput improvement observed on the CXL-S architecture can be attributed to Fanmem's core design principles. The increased latency from the switch to the memory server allows Fanmem to effectively hide more latency, resulting in improved performance.

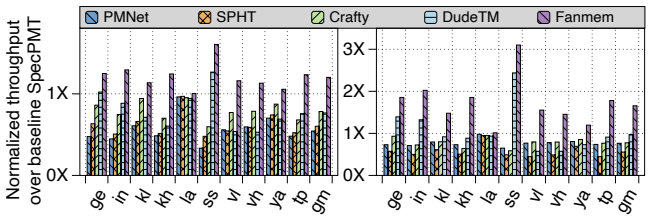

Figure 11. Normalized throughput over the baseline on CXL-F (left) and CXL-S (right) architectures. *gm* denotes geomean.

Fanmem achieves higher throughput in applications with small working sets and short transactions. For instance, *labyrinth*, which has the largest working set (433 and 458 cache lines for read and write) and the highest number of instructions (219,571 instructions) per transaction, results in a negligible proportion of log persistence time relative to total execution time. Consequently, Fanmem's optimization of log persistence has minimal impact on its performance. In contrast, *ssca2* has the smallest working set (10 and 4 cache lines for read and write) and the lowest number of instructions (50 instructions) per transaction, resulting in a relatively long logging time and a maximum throughput of 3.1× over the baseline. *genome*, similar to *intruder*, has a medium-sized working set (98 and 15 cache lines for read and write) and slightly more instructions (1,717 instructions) than *ssca2*. As a result, the logging time is lower compared to *ssca2*, leading to an average acceleration effect.

The asynchronous transaction DudeTM maintains a partial copy of disaggregated memory in local DRAM. When

the system encounters data not present in the local DRAM, it triggers a page fault, copying the page from disaggregated memory to the local DRAM, causing serious write amplification and resulting in poor performance.

Fanmem considerably outperforms all other schemes, as shown in Figure 11. On average, it achieves 1.5× and 2.1× higher throughput over Crafty on two CXL platforms, 2.0× and 3.0× over SPHT, and 1.6× and 1.7× over DudeTM.

To further analyze the sources of Fanmem's performance gains, Figure 12 presents a breakdown of the transaction execution time (i.e., from `tx_begin()` to `tx_end()`) for each benchmark, comparing Fanmem against the SpecPMT baseline. By default, Fanmem performs polling at the beginning and end of each transaction to monitor persistence progress and commit transactions when ready. This breakdown distinguishes the time spent on core transaction logic from the overheads introduced by supporting operations such as logging and polling. The figure shows that SpecPMT incurs a majority (53.3%) of its execution time on logging. Thanks to its asynchronous persistence, Fanmem reduces the logging time by 71%, while adding a polling overhead of only 0.7%.

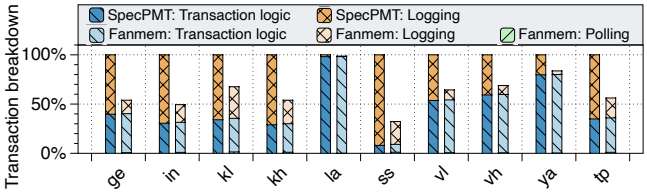

Figure 12. Transaction execution breakdown on the CXL-S configuration. Polling incurs negligible overhead.

6.3 Scalability

Fanmem demonstrates clear throughput superiority over traditional persistent memory transaction schemes, especially in small-scale settings in TPC-C. Figure 13 illustrates the throughput of various schemes. The figure shows that apart from Fanmem and SpecPMT, the throughput of other schemes stalls at 16 or more threads. Fanmem's throughput continues to increase to 32 threads, where it achieves 1.4× to 5.3× higher throughput compared to other schemes.

The degree of write amplification negatively correlates with scalability, indicating it is a major factor. For example, despite DudeTM's reliance on asynchronous transactions to reduce log persistence overhead, it incurs 4–6× write amplification by maintaining duplicate copies of both data and logs in its shadow memory. SPHT also relies on shadow memory and replays logs to persistent memory, and Crafty avoids shadow memory overhead through re-execution, but they require persisting both logs and data, resulting in at least 2× write amplification. Fanmem and SpecPMT do not maintain shadow memory or data copies, hence their write amplification is low, allowing them to be scalable. Furthermore, Fanmem forwards log write requests non-blockingly via the CXL switch, hence it improves throughput over SpecPMT.

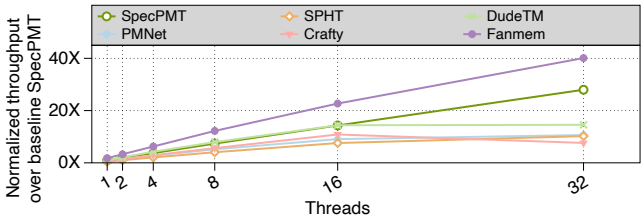

Figure 13. Transaction throughput under CXL-S configuration as the number of threads increases. Normalized to single-threaded SpecPMT.

Fanmem's architecture is inherently extensible. For heavier transactional loads, the system can scale out by deploying additional Fanmem-enabled CXL switches. This modularity is critical for maintaining high throughput in large-scale disaggregated environments. Another option is to use the dynamic fallback mechanism (Section 4.5) under contention.

6.4 Transaction Schemes

We apply Fanmem's acceleration on SPHT, demonstrating its applicability. As shown in Figure 14, Fanmem-SPHT outperforms the vanilla SPHT by 1.3× and 1.8× on the two CXL configurations. Additionally, Fanmem achieves an average throughput of 1.6× and a maximum throughput of 3.0× over Fanmem-SPHT, suggesting that speculative logging is a more performant choice when combined with Fanmem.

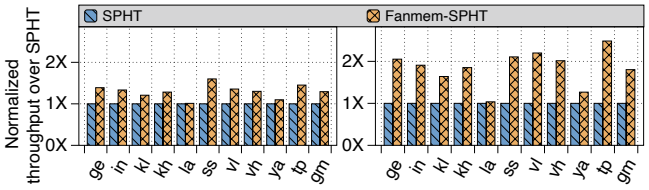

Figure 14. The throughput of Fanmem-accelerated SPHT (Fanmem-SPHT) compared to SPHT on CXL-F (left) and CXL-S (right) architectures. *gm* denotes geomean.

6.5 Sensitivity Analysis

6.5.1 Latency between CXL Switch and Memory Server.

Fanmem achieves higher throughput when the latency between the switch and memory server increases, as the baseline solution devotes more execution time to flushing data to the disaggregated memory in such scenarios. As shown in Figure 15, Fanmem achieves 1.1× to 2.0× higher throughput than the baseline, SpecPMT, on average when latency increases from 45ns to 745ns. The throughput increases largely linearly with the increase in latency, highlighting Fanmem's

efficiency and trend in reducing the impact of latency on overall performance.

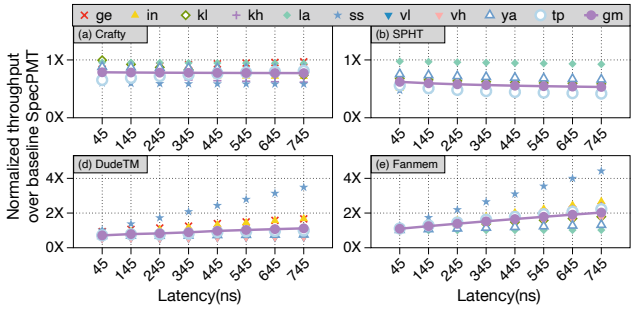

Figure 15. Throughput sensitivity to the latency between the CXL switch and the memory server. *gm* denotes geomean.

Crafty and SPHT achieve stable throughput over the baseline when the latency increases. In contrast, the asynchronous schemes, Fanmem and DudeTM, effectively hide the latency and achieve higher throughput when the latency rises from 45ns to 745ns. For instance, DudeTM's throughput improves from 0.7× to 1.1× over this latency range.

The applications can be categorized into two groups based on their sensitivity to the latency between the switch and memory server. The first group, including *labyrinth* and *yada*, is less sensitive to latency changes, with speedup variations of less than 30% when using Fanmem. These applications have large working sets and long transactions, and variations in latency have a minimal impact on the overall execution time. The second group, exemplified by *ssca2*, exhibits a noticeable throughput increase of 315% with Fanmem. For instance, Fanmem achieves 1.3× higher throughput on *ssca2* with 45ns latency between the switch and memory server, which increases to 4.4× with 745ns latency. This application devotes more time to log persistence due to its small working set and short transactions, providing ample opportunity for Fanmem to hide the persistence latency.

6.5.2 Latency between Compute Server and CXL Switch. The performance gains of all transaction schemes are largely stable because both the baseline and the other schemes suffer from similar performance losses when latency increases. Fanmem's performance gain ranges from 1.7× to 1.6×, despite the latency between the compute server and CXL switch varies from 40ns to 160ns, as illustrated in Figure 16 (e). Fanmem's performance declines as latency between the compute server and the CXL switch increases. This insight suggests that it is beneficial to position the CXL switch close to the compute server. In data centers, an optimal placement of CXL switches is on the same rack as the compute server, minimizing latency and maximizing performance.

Some applications, such as *yada*, are less sensitive to the latency between the compute server and CXL switch. This

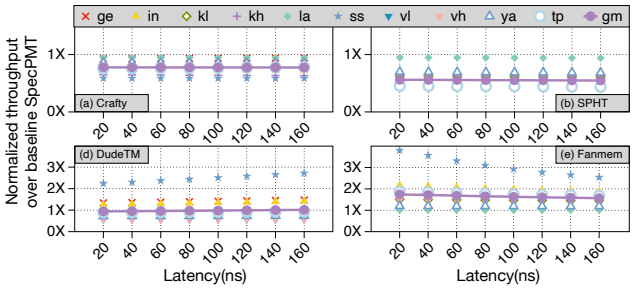

Figure 16. Throughput sensitivity to the latency between the compute server and the CXL switch. The performance gain is more pronounced with a low-latency interconnection. *gm* denotes geomean.

insensitivity occurs because they devote most of their execution time to computation rather than log persistence.

However, the speedup can substantially drop when the latency increases. For example, Fanmem accelerates *ssca2* by 3.8× with the lowest latency but only by 2.5× with the highest latency. This occurs because *ssca2* devotes most of its execution time to log persistence, and the increased latency adds overhead when sending the log to the switch.

6.5.3 Logging Pressure. To evaluate the performance sensitivity to logging pressure intensity, Figure 17 shows Fanmem's improvement over SpecPMT configured with varying identical log sizes. Expanding the log size from 1× to 8× increases the execution time of both schemes, but Fanmem's more efficient logging allows it to increase its throughput over SpecPMT (from 1.7× to 2.2×) as the log size increases. This result highlights the increasing importance of Fanmem's optimizations as log sizes get larger.

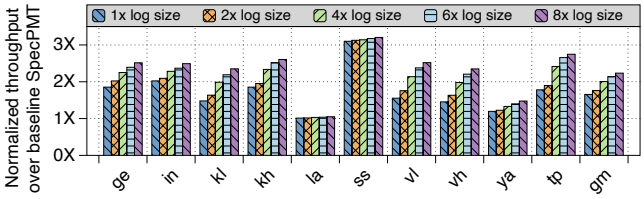

Figure 17. Throughput sensitivity to different log sizes on the CXL-S architecture. *gm* denotes geomean.

6.5.4 pLAT and pLWST Size. The sizes of pLAT and pLWST constrain the number of concurrent threads utilizing Fanmem's optimizations. If threads exceed these entries, Fanmem falls back to the default software-only logging scheme, resulting in smaller performance gains. Thus, ensuring sufficient pLAT and pLWST sizes is important to maximize Fanmem's benefits. Figure 18 shows that as both structures scale from 0 to 32 entries, Fanmem's throughput relative to the baseline SpecPMT increases from 1.0× to 1.4×. With 0 entries, Fanmem achieves identical performance to SpecPMT.

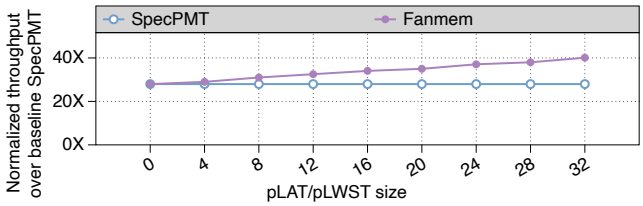

Figure 18. Throughput sensitivity to the size of pLAT and pLWST on the CXL-S configuration. Normalized to single-threaded SpecPMT.

6.5.5 Types of Memory. As illustrated in Figure 19, Fanmem demonstrates substantial throughput improvements across all memory technologies, though with varying degrees of enhancement. As memory access latency increases from UPS-backed DRAM to PM to CMM-H, the relative benefit of eliminating persistency operations from the critical path diminishes, as the performance bottleneck shifts from persistency operations to the inherent memory technology latency. Specifically, Fanmem achieves throughput improvements of 65.6% on UPS-backed DRAM servers, 59.3% on PM servers, and 43.6% on CMM-H servers compared to the baseline SpecPMT with the same memory technologies.

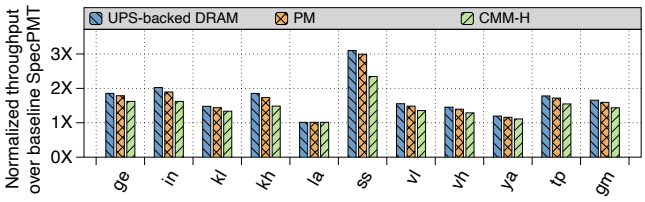

Figure 19. Throughput sensitivity to different memory technologies on the CXL-S architecture. *gm* denotes geomean.

6.6 Commit Latency Breakdown

Figure 20 illustrates how the polling frequency affects transaction commit latency and decomposes this latency into five components: polling, transaction logic, log offloading, log persistence, and the waiting period from persistence completion to the subsequent polling event. By default, Fanmem polls persistence progress twice per transaction at `tx_begin()` and `tx_end()` respectively. Under this configuration, polling constitutes only 0.5% of total commit latency, while transaction logic, log offloading, and log persistence account for 24.6%, 10.5%, and 25.3%, respectively. The remaining 39.1% represents the waiting period. Increasing polling frequency from two to six polls per transaction by inserting more polling invocations in the transactions raises polling overhead from 0.5% to 1.5%, yet reduces overall commit latency by 25.6%. This reduction stems primarily from a 67.3% decrease in waiting period, demonstrating that more frequent polling substantially mitigates the latency between persistence completion and commit confirmation, thereby lowering overall commit latency. However, excessive polling may consume substantial CXL bandwidth and CPU resources, potentially causing the latency introduced by polling to outweigh the latency benefits achieved.

As thread count increases, transaction commit latency grows due to thread contention, which consequently reduces polling's relative proportion. Simultaneously, higher thread counts amplify polling frequency, thereby reducing the waiting period for commit. For example, compared to single-threaded execution, 32-threaded TPC-C exhibits a polling overhead reduction from 0.8% to 0.6%, while the waiting period shows a dramatic decrease from 20.9% to 0.5%.

6.7 Overhead over Logless Execution

While state-of-the-art transaction schemes such as Crafty and SPHT incur considerable overhead (3.4× and 4.8× on average), Fanmem mitigates most of the overhead, resulting in only a 60.1% increase in execution time for logging and persistence, as shown in Figure 21.

6.8 Recovery

The recovery progress of Fanmem is mostly the same as that of SpecPMT and SPHT, with additional timestamp checks. We inject random failures according to prior work [59]. During recovery, the system retrieves the most recent per-process timestamp from the memory server and replays all committed log entries whose timestamps precede this value, thereby restoring consistent states on the memory server. The timestamp comparison introduces modest overhead, causing Fanmem to exhibit 6.4% and 2.5% longer recovery times compared to SPHT and SpecPMT, respectively. Since system failures occur infrequently and recovery executes outside the critical path of normal program execution, this minimal overhead does not impact overall system performance. Moreover, the recovery process supports parallelization by replaying independent logs concurrently, which can further reduce recovery latency.

6.9 Hardware Resource and Power Consumption

The entire design of Fanmem requires a modest 2,590 *Flip-Flops* (FFs), 3,361 *Look-Up Tables* (LUTs), and 594KB of on-chip *Block RAM* (BRAM), corresponding to just 0.1%, 0.2%, and 8.5% of the total resources available on the U250 FPGA, respectively. LAT utilizes 1,475 FFs, 1,949 LUTs, 269KB BRAM, and requires 6 cycles for lookup, while LWST utilizes 673 FFs, 1,056 LUTs, 325KB BRAM, and requires 2-3 cycles for update. To place these figures in context, we compare Fanmem's resource footprint with that of MiCache [60], a representative memory-side cache. Despite a comparable BRAM utilization (106.5%), Fanmem consumes less logic, using only 86.3% of the FFs and 54.2% of the LUTs required by MiCache. This comparison highlights the lightweight nature of Fanmem relative to a full-fledged caching system.

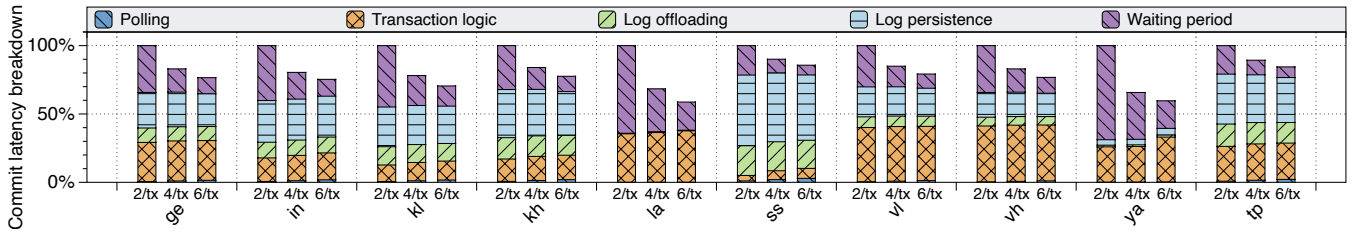

Figure 20. Commit latency breakdown on the CXL-S configuration. 2/tx, 4/tx, and 6/tx represent the polling frequency (poll 2, 4, 6 times per transaction on average).

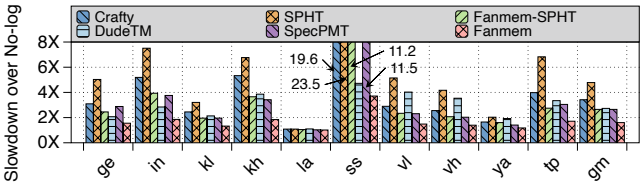

Figure 21. Slowdown of various transaction schemes relative to a No-log baseline on the CXL-S configuration. Lower is better. *gm* denotes geomean.

Consistent with prior works [34, 63, 64], we use CACTI 7.0 [9] to estimate the silicon area and the power consumption of the core components of an ASIC implementation of Fanmem. Fanmem requires 1.4 mm^2 of silicon area and 479.2 mW of static power under a 32 nm process technology. The LAT uses 231 mW with 1.5 ns access latency, while the LWST consumes 248.2 mW with 1.1 ns access latency.

7 Related Work

7.1 Persistent Memory Transaction Systems

The most relevant field to this work is persistent memory transactions. We summarize them into two categories: software and hardware persistent memory transactions. Table 3 lists representative software and hardware solutions.

Table 3. Comparison of different schemes

Schemes	CPU Independent	Reduce CPU stall	Scalability
SpecPMT [66]	Yes	No	High
Crafty [19]	Yes	No	Low
SPHT [14]	Yes	No	Low
DudeTM [39]	Yes	Yes	Low
PMNet [48]	Yes	Yes	Low
ASAP [6]	No	Yes	High
HOOP [13]	No	No	High
Fanmem	Yes	Yes	High

7.1.1 Software Persistent Memory Transactions. Researchers have proposed optimizations for software persistent memory transactions, which fall into three categories.

The first category employs asynchronous persistence to remove persistency from the critical path [14, 30, 39, 42, 43, 48]. DudeTM [39] and SPHT [14] use out-of-place updates with background persistence, but address redirection incurs significant overhead [31, 66]. Kamino-Tx [42] achieves in-place updates by tracking dependencies, yet access snooping causes substantial slowdown. PMNet [48] leverages NIC-side persistent memory but suffers from high write overhead.

The second category reduces data persistence and ordering overhead [16, 23, 43, 59, 66]. Mnemosyne [55] reduces ordering instructions for redo logging. LSNVMM [23] employs a log-structured tree with hash-based indexing. Pronto [43] uses semantic logging to eliminate data persistence. SpecPMT [66] uses logs for data recovery, eliminating data persistence and reducing memory ordering.

The third category employs re-execution [7, 19, 24, 40, 43, 61]. Crafty [19] uses hardware transactional memory [15] to re-execute write operations. Pronto [43] records operations for checkpoint-based restoration. iDO [40], JUSTDO [24], LP [7], and Clobber-NVM [61] rely on specific scenarios or exploit idempotence, limiting their applicability.

These software methods are not designed for disaggregated scenarios and cannot significantly reduce persistence overhead in disaggregated memory systems. Fanmem eliminates most of the overhead via switch-supported operations.

7.1.2 Hardware Persistent Memory Transactions. Hardware schemes leverage hardware to achieve fine-grained optimizations. While these solutions achieve promising performance, they all require CPU modifications, making widespread adoption difficult. In contrast, Fanmem offloads log persistency to the CXL switch, avoiding any CPU modifications. Existing works fall into the following categories.

The first category employs hardware asynchronous persistency [6, 13, 17, 45]. For example, HOOP [13] offloads logging to the memory controller, and implements out-of-place data modification and access redirection using an on-chip address translation table. ASAP [6] determines the dependencies between log and data, leading to significant area cost (2.5% of CPU area).

The second category reduces write overhead to improve performance by minimizing unnecessary writes [26, 47, 50, 58, 65, 70] or coalescing log writes [26, 28, 65]. For example,

ATOM [28] and Silo [70] reduce the write overhead by merging log entries. SLPMT [65] combines selective logging and hardware persistent memory transactions to reduce logging overhead, while further minimizing write overhead through log merging and batching.

The third category reduces ordering overhead using fine-grained barriers or hardware-enforced ordering without explicit instructions [25, 28, 45, 49–51]. For instance, EDE [51] introduces instructions expressing log-data persistence constraints, allowing hardware reordering without store fences.

The fourth category employs a logless approach [21, 72]. Kiln [72] and LAD [21] use on-chip persistent buffers to hold updates and apply them to persistent memory when the transaction commits, avoiding the use of logs and reducing memory write traffic. However, these logless approaches require CPU modifications.

7.2 Whole System Persistence

Recent research has proposed mechanisms for *Whole System Persistence* (WSP) for crash consistency [22, 27, 67, 69, 73]. PPA [67] replays unpersisted stores committed before failure. LightWSP [73] uses recoverable regions and achieves persistence using a non-temporal data path. cWSP [69] partitions programs into idempotent regions for re-execution. Capri [27] employs compiler-architecture co-design for efficient recovery. Zhuque [22] preserves volatile architectural state to restart execution after failure. While WSP guarantees data consistency at the system level, it requires special CPU architecture support.

7.3 Disaggregated Memory Reliability

Recent works addressing disaggregated memory reliability include memory server redundancy [33], failure-resilient memory management [71], and persistence extension into network [48]. Most existing approaches safeguard specific failure domains, e.g., memory server crash or network disconnect, rather than providing a unified transaction model resilient to compute, network, and memory server failures. Fanmem's design uniquely ensures failure-atomic crash consistency for all key components (compute server, switch, and memory server) by maintaining logs within the remote memory domain.

8 Conclusion

We propose Fanmem, the first failure-atomic transactions designed specifically for CXL-based memory disaggregated architectures. Fanmem provides efficient crash consistency upon failures in compute servers, CXL switches, and memory servers. The system achieves satisfactory performance by efficiently offloading the time-consuming data persistence to the CXL switch, effectively moving the associated latency off the critical path of transaction execution. The evaluation confirms Fanmem's efficiency on two representative CXL platforms.

Acknowledgments

We express our gratitude to the ASPLOS reviewers for providing valuable and constructive comments. This work is supported by the National Key Research and Development Program of China under grant 2022YFB4501400, the National Natural Science Foundation of China (NSFC) under grant 62572207 and 62372199, the NSFC-RGC under grant 62461160333, the National Science Foundation (NSF) under grants CNS-2107068, CNS-2312207, 2106629, and 2312206, Wuxue Minben Mineral Resources Development Co., Ltd, and Xiaomi Young Talents Program. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] 1992. TPC-C Benchmark. https://www.tpc.org/tpcc/.
[2] 2022. *3.3.2 CXL.mem Channel Description* (version 1.0 ed.). Vol. Revision 3.0.
[3] 2022. Intel libvmmalloc. https://pmem.io/vmem/libvmmalloc/.
[4] 2023. With a systems approach to chips, Microsoft aims to tailor everything "from silicon to service" to meet AI demand. https://news.microsoft.com/source/features/ai/in-house-chips-silicon-to-service-to-meet-ai-demand/.
[5] 2023. Xilinx Alveo U250. https://www.amd.com/en/products/accelerators/alveo/u250/a-u250-a64g-pq-g.html.
[6] Ahmed Abulila, Izzat El Hajj, Myoungsoo Jung, and Nam Sung Kim. 2022. ASAP: Architecture Support for Asynchronous Persistence. In *Proceedings of the 2022 ACM/IEEE 49th Annual International Symposium on Computer Architecture*. 306–319.
[7] Mohammad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy Persistency: A High-Performing and Write-Efficient Software Persistency Technique. In *Proceedings of the 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*. 439–451.
[8] Amazon. 2023. *AWS Graviton Processors*. https://aws.amazon.com/ec2/graviton/.
[9] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization* 14, 2 (2017), 1–25.
[10] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. 2023. Design Tradeoffs in CXL-Based Memory Pools for Public Cloud Platforms. *IEEE Micro* 43, 2 (2023), 30–38.
[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathiiit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.
[12] Tony Brewer and Nathan Kalyanasundharam. 2022. CXL3 Fabric: Introduction and Use Cases. https://hc34.hotchips.org/.
[13] Miao Cai, Chance C. Coats, and Jian Huang. 2020. HOOP: Efficient Hardware-Assisted Out-of-Place Update for Non-Volatile Memory. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. 584–596.

[14] Daniel Castro, Alexandro Baldassin, João Barreto, and Paolo Romano. 2021. SPHT: Scalable Persistent Hardware Transactions. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. 155–169.

[15] Daniel Castro, Paolo Romano, and João Barreto. 2019. Hardware Transactional Memory Meets Memory Persistency. *Journal of Parallel and Distributed Computing* 130 (2019), 63–79.

[16] Joel Coburn, Adrian M. 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 *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 105–118.

[17] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas, and Stefanos Kaxiras. 2021. TSOPER: Efficient Coherence-Based Strict Persistency. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture*. 125–138.

[18] AL-Hazemi Fawaz, Josip Lorincz, and Alaelddin FY Mohammed. 2019. Minimizing Data Center Uninterruptable Power Supply Overload by Server Power Capping. *IEEE Communications Letters* 23, 8 (2019), 1342–1346.

[19] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-compatible Persistent Transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 59–74.

[20] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *Proceedings of the 2022 USENIX Annual Technical Conference*. 287–294.

[21] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. 2019. Distributed Logless Atomic Durability with Persistent Memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 466–478.

[22] George Hodgkins, Yi Xu, Steven Swanson, and Joseph Izraelevitz. 2023. Zhuque: Failure is Not an Option, it's an Exception. In *Proceedings of the 2023 USENIX Annual Technical Conference*. 833–849.

[23] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *Proceedings of the 2017 USENIX Annual Technical Conference*. 703–717.

[24] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. 427–442.

[25] Jungi Jeong and Changhee Jung. 2021. PMEM-Spec: Persistent Memory Speculation (Strict Persistency Can Trump Relaxed Persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 517–529.

[26] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient Hardware-Assisted Logging with Asynchronous and Direct-Update for Persistent Memory. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 520–532.

[27] Jungi Jeong, Jianping Zeng, and Changhee Jung. 2022. Capri: Compiler and Architecture Support for Whole-System Persistence. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 71–83.

[28] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-Volatile Memory through Hardware Logging. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture*. 361–372.

[29] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture*. 66–78.

[30] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-performance Transactions for Persistent Memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. 399–411.

[31] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 335–349.

[32] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. 2023. Cache in hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. 24–30.

[33] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. 2022. Hydra: Resilient and Highly Available Remote Memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies*. 181–198.

[34] Chuanhan Li, Jishen Zhao, and Yuanchao Xu. 2025. Efficient Security Support for CXL Memory through Adaptive Incremental Offloaded (Re-)Encryption. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. 1102–1116.

[35] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 574–587.

[36] Jianing Li, Jiabin Liu, Xingyuan Hu, Yuhang Zhang, Guosheng Yu, Shimeng Qian, Wei Mao, Li Du, Yongfu Li, and Yuan Du. 2023. Grand Challenge on Software and Hardware Co-Optimization for E-Commerce Recommendation System. In *Proceedings of the 2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems*. 1–5.

[37] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1203–1217.

[38] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. 2019. Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2223–2236.

[39] 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. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. 329–343.

[40] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 258–270.

[41] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 742–755.

[42] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *Proceedings of the 12th European Conference on Computer Systems*. 499–512.

[43] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 789–806.

[44] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*. 35–46.

[45] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*. 336–349.

[46] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and H. Peter Hofstee. 2020. ThymesisFlow: A Software-Defined, HW/SW co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 868–880.

[47] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*. 672–685.

[48] Korakit Seemakhupt, Sihang Liu, Yasas Senevirathne, Muhammad Shahbaz, and Samira Khan. 2021. PMNet: In-network Data Persistence. In *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*. 804–817.

[49] Sara Mahdizadeh Shahri, Seyed Armin Vakil Ghahani, and Aasheesh Kolli. 2020. (Almost) Fence-Less Persist Ordering. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 539–554.

[50] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 178–190.

[51] Thomas Shull, Ilias Vougioukas, Nikos Nikoleris, Wendy Elsasser, and Josep Torrellas. 2021. Execution Dependence Extension (EDE): ISA Support for Eliminating Fences. In *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*. 456–469.

[52] Rachee Singh, Muqeet Mukhtar, Ashay Krishna, Aniruddha Parkhi, Jitendra Padhye, and David Maltz. 2021. Surviving Switch Failures in Cloud Datacenters. *ACM SIGCOMM Computer Communication Review* 51, 2 (2021), 2–9.

[53] Mohammadreza Soltaniyeh, Gongjin Sun, Xuebin Yao, Amir Beygi, Ramdas Kachare, Dongwan Zhao, Hingkwan Huen, Andrew Chang, Senthil Murugesapandian, and Caroline Kahn. 2025. Revisiting Memory Hierarchies with CMM-H: Use Device-Side Caching to Integrate DRAM and SSD for a Hybrid CXL Memory. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems*. 45–51.

[54] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 105–121.

[55] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 91–104.

[56] Luming Wang, Xu Zhang, Songyue Wang, Zhuolun Jiang, Tianyue Lu, Mingyu Chen, Siwei Luo, and Keji Huang. 2024. Asynchronous Memory Access Unit: Exploiting Massive Parallelism for Far Memory Access. *ACM Transactions on Architecture and Code Optimization* 21, 3 (2024), 55:1–55:28.

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

[58] Xueliang Wei, Dan Feng, Wei Tong, Jingning Liu, and Liuqing Ye. 2020. Morlog: Morphable Hardware Logging for Atomic Persistence in Non-Volatile Main Memory. In *Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*. 610–623.

[59] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. 2021. ArchTM:Architecture-Aware, High Performance Transaction for Persistent Memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. 141–153.

[60] Shaoxian Xu, Sitong Lu, Zhiyuan Shao, Xiaofei Liao, and Hai Jin. 2024. MiCache: An MSHR-Inclusive Non-Blocking Cache Design for FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 22–32.

[61] Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, re-Execute More. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 346–359.

[62] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. 2022. Temporal Exposure Reduction Protection for Persistent Memory. In *Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture*. 908–924.

[63] Sujay Yadalam, Nisarg Shah, Xiangyao Yu, and Michael Swift. 2022. ASAP: A Speculative Approach to Persistence. In *Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture*. 892–907.

[64] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Xiaofei Liao, Hai Jin, and Yan Solihin. 2021. Supporting Legacy Libraries on Non-Volatile Memory: A User-Transparent Approach. In *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture*. 443–455.

[65] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Yan Sha, Xiaofei Liao, Hai Jin, and Yan Solihin. 2023. Reconciling Selective Logging and Hardware Persistent Memory Transaction. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture*. 664–676.

[66] Chencheng Ye, Yuanchao Xu, Xipeng Shen, Yan Sha, Xiaofei Liao, Hai Jin, and Yan Solihin. 2023. SpecPMT: Speculative Logging for Resolving Crash Consistency Overhead of Persistent Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 762–777.

[67] Jianping Zeng, Jungi Jeong, and Changhee Jung. 2023. Persistent Processor Architecture. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 1075–1091.

[68] Jianping Zeng, Shuyi Pei, Da Zhang, Yuchen Zhou, Amir Beygi, Xuebin Yao, Ramdas Kachare, Tong Zhang, Zongwang Li, Marie Nguyen, Rekha Pitchumani, Yangsoek Ki, and Changhee Jung. 2025. Performance Characterizations and Usage Guidelines of Samsung CXL Memory Module Hybrid Prototype. *arXiv preprint arXiv:2503.22017* (2025).

[69] Jianping Zeng, Tong Zhang, and Changhee Jung. 2024. Compiler-Directed Whole-System Persistence. In *Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture*. 961–977.

[70] Ming Zhang and Yu Hua. 2023. Silo: Speculative Hardware Logging for Atomic Durability in Persistent Memory. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture*. 651–663.

[71] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. 2023. Partial Failure Resilient Memory Management System for (CXL-Based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 658–674.

[72] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 421–432.

[73] Yuchen Zhou, Jianping Zeng, and Changhee Jung. 2024. Lightwsp: Whole-System Persistence on the Cheap. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*. 215–230.