Hierarchical Framework for Multi-node Compute eXpress Link Memory Transactions

Ellis Giles
egiles@acm.org
Elex Technologies
Houston, TX, USA

Peter Varman
pjv@rice.edu
Rice University
Houston, TX, USA

ABSTRACT

There is a growing need to support high-volume, concurrent transaction processing on shared data in both high-performance and datacenter computing. A recent innovation in server architectures is the use of disaggregated memory organizations based on the Compute eXpress Link (CXL) interconnect protocol. While CXL memory architectures alleviate many concerns in datacenters, enforcing ACID semantics for transactions in CXL memory faces many challenges.

We describe a novel solution for supporting ACID (Atomicity, Consistency, Isolation, Durability) transactions in a CXL-based disaggregated shared-memory architecture. We call this solution HTCXL for Hierarchical Transactional CXL. HTCXL is implemented in a software library that enforces transaction semantics within a host along with a back-end controller to detect conflicts across hosts. HTCXL is a modular solution allowing different combinations of HTM or software-based transaction management to be mixed as needed.

We perform experimental evaluation of HTCXL using micro-architectural processor simulation and several STAMP benchmarks. Our method shows a significant speedup over a software approach on CXL fabric.

CCS CONCEPTS

• **Computer systems organization** → **Processors and memory architectures**; • **Computing methodologies** → *Concurrent computing methodologies*; • **Information systems** → *Storage class memory*.

1 INTRODUCTION

Recent years have witnessed a sharp shift towards real time data-driven and high-throughput applications. This shift has spurred a broad adoption of in-memory and massively parallelized data processing. across business, scientific, and industrial application domains Most recently, Artificial Intelligence (AI), Machine Learning (ML) based applications have exploded in popularity, pushing the limits of memory performance. Applications in these cutting-edge domains require access to large volumes of low-latency, high-throughput data coupled with commensurate computational power for data operations. Fortunately, hardware developments like Compute eXpress Link™ (CXL™) [8], an open standard for a cache-coherent interconnect for processors, accelerators, and heterogenous memory, have the potential to support a scalable distributed processing infrastructure, providing high-bandwidth access to large shared memory pools with reasonable latencies.

Emerging CXL-based memory architectures create new opportunities and challenges for managing concurrency in multi-threaded, high-throughput, data-sharing applications. A common approach to handle the complexity of concurrency control is to structure the critical sections as *transactions*. Transactions provide ACI (Atomicity, Consistency and Isolation) guarantees to memory-resident (volatile) data structures, making it easier to program and verify the correctness of in-memory concurrent applications. By adding Durability requirements to a transaction, crash-resistant operation can be supported using non-volatile memory. The responsibility for providing transaction support is delegated to the processor hardware or system software.

This paper describes a novel technique for transaction management in CXL-supported disaggregated memory architectures, for both partitioned and shared memory organizations [8]. It supports volatile transactions as well as persistent ACID transactions. An earlier paper [55] proposed hardware augmentation to support unbounded-size transactions in a Hardware Transaction Memory (HTM)-enabled host connected to a pooled CXL memory subsystem. Another work [19] describes a method to support small transactions across multiple HTM -enabled hosts sharing CXL memory with contention resolution at the CXL controller. This paper describes a hierarchical approach where transaction management within the host and the CXL controller work synergistically using a simple contention resolution algorithm. Our approach does not rely on HTM support at the host, can handle unbounded transaction sizes, and is sensitive to the latencies of CXL access by avoiding frequent fine-grained synchronous memory accesses.

Our paper makes the following contributions:

- We introduce Hierarchical Transactional CXL (HTCXL) which adds ACID support to memory based transactions over CXL. Our approach introduces no changes to processor, cache, or cache-coherency mechanisms, relying solely on software techniques and back-end processing at the controller.
- HTCXL decouples Atomicity, Isolation, and Durability into independent components that are managed synergistically by host-managed software and the CXL controller.
- HTCXL supports and inter-operates with both HTM and non-HTM based transactions, retaining the desirable properties of existing HTM implementations.
- We evaluate HTCXL using a cycle-accurate simulator and multiple microbenchmarks and transactional benchmarks from the STAMP suite. We demonstrate that HTCXL continues to scale well across multiple nodes in the CXL fabric, achieving significant improvements over an STM approach.

The remainder of the paper is organized as follows. Section 2 discusses CXL and TM (Transaction Management) technologies. An overview of our solution (HTCXL) for hierarchical transaction management in CXL is presented in Section 3. Section 4 describes the

HTCXLimplementation: intra-node processing and the operations of the CXL controller are described in Sections 4.1 and 4.2 respectively. Section 5 presents evaluation results. Related work is discussed in Section 6 and the paper is summarized in Section 7.

2 BACKGROUND

In this section, we provide an overview of Compute eXpress Link (CXL) and Transaction Memory (TM) implementation technologies. We summarize previous work in these areas and compare them with our proposal.

2.1 Compute eXpress Link (CXL)

CXL (Compute Express Link) is a cache-coherent interconnect protocol that facilitates sharing and reduces software stack complexity [8]. The recent CXL 3.1 Specification [45] adds Global Fabric Attached Memory (GFAM) functionality that enables sharing between distributed nodes (hosts, accelerators, CXL memory pools).

Two proposed use cases of the CXL interconnect are *Memory Pooling* and *Memory Sharing*. In the former, a remote pool of (possibly heterogenous) memory devices are accessed by distributed servers through a CXL link. The memory pool is divided among the hosts based on their anticipated memory requirements. At any time a partition is dedicated to a single host although coarse-grained memory repartitioning can be performed if desired. In *Memory Sharing*, the applications running on the hosts share a common pool of memory, logically akin to multiple cores (sockets) sharing system DRAM (NUMA). While the specification includes hardware-managed cache-coherence across the CXL fabric, system-wide implementation and performance issues in a distributed shared-memory model are currently unknown.

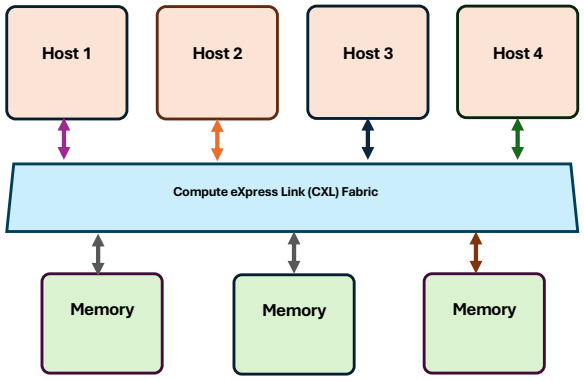


Figure 1: Disaggregated CXL Memory Architecture. In *Memory Pooling* CXL memory is partitioned among the hosts. With *Memory Sharing* hosts can share CXL memory.

2.2 Transaction Memory (TM)

A transaction is a unit of execution that satisfies ACID properties: Atomicity (either all or none of a transaction's updates are reflected in transaction memory), Consistency (an application-dependent set of invariants that must hold before and after transaction execution), Isolation (the updates of a transaction are not visible to other transactions till it completes), and, optionally, Durability (the state of a committed transaction must survive power failure by being persisted to non-volatile storage).

Transaction semantics can be enforced using either hardware or software mechanisms. HTM is a mechanism available in high-performance processors from Intel [22], ARM [1], and IBM [29], which provide hardware support for memory based transactions satisfying ACI semantics on a cache-coherent multi-core processor. For instance, in Intel's Restricted Transactional Memory (RTM) [23, 24] transaction code sections (demarcated by **begin_HTM** and **end_HTM** instructions) can execute concurrently on different cores. Hardware monitors the transactions and uses the L1 cache coherence mechanism to detect read/write conflicts. A write (read) by a transaction to a transaction variable held in any state (modified state) in another core's L1 cache will cause one of the transactions to abort. Transaction variables are buffered and pinned to the L1 cache for the duration of the transaction. On transaction commit, they are atomically made visible to all cores and released for eviction to memory. Transactions must be aborted if there are capacity or conflict misses in the L1 cache, limiting their size. Proposals for overcoming the size restriction using software interception and logging of overflow variables have been proposed for in-memory transactions in [51] and for persistent memory transactions in [25, 26], while a hardware-based scheme using a victim cache for both memory and durable transactions in CXL memory was presented in [55].

Software Transaction memory (STM) employs software intervention to maintain ACI semantics for transactions within a host. Most implementations require extensive locking or version maintenance leading to low performance. An extensive set of approaches are described in [21]. Our software technique uses ideas from the HTM implementation to allow for a simple and fast implementation, which can be replicated at different levels of the memory hierarchy. Furthermore, our hierarchical approach is modular and an unbounded transaction HTM implementation could be substituted for out software approach if appropriate hardware were available.

3 OVERVIEW

We introduce Hierarchical Transactional CXL or HTCXL, which supports ACID requirements for transactions running on hosts connected to remote shared memory over a CXL interface. A transaction consists of a single sequential thread that runs on a core and accesses memory variables stored in shared CXL memory. Multiple transactions that may share memory variables run concurrently on cores on the same or different nodes.

HTCXL is a two-level solution: a local concurrency control protocol enforces ACI transaction semantics between transactions running on a host, while conflicting transactions running on different nodes are serialized by a similar protocol running at the CXL controller. If durability is desired the protocol at the CXL controller can be easily extended to persist transactions atomically on non-volatile media at the CXL node. without altering the rest of the protocol.

3.1 Intra-node Transaction Management

The mechanisms within a node provide *atomicity* and *isolation* between the threads running on the cores. Since a transaction may

abort during its execution, updates made prior to the abort could leave the memory in an inconsistent state; the system should ensure that either all or none of a transaction's updates are reflected in the CXL memory. Isolation hides the updates made by in-progress transactions so that they are not visible to other transactions. Ignoring isolation could lead to a violation of the fundamental serializability requirement of transactional execution. For instance, if two transactions both read variables updated by the other transaction, it implies a circular dependence preventing serializability. We discuss both these issues below.

Atomicity requires that updates made by a thread are treated as speculative until the transaction completes. One approach is to make all transaction updates to temporary memory locations, and update CXL memory only on transaction commit[1]. In high-performance processors with hardware support for Transaction Memory (HTM) [1, 22, 29], updates are held in the cache hierarchy or in write buffers until a transaction commits. In Intel's Restricted Transaction Memory (RTM) [23, 24], for instance, all updates are restricted to the core's L1 cache, and cache conflicts that overflow the cache cause the transaction to abort. A mechanism to handle the cache overflow of transaction variables using a hardware structure called a transaction victim cache was proposed in [11, 14, 40, 55], while software techniques based on memory logging of overflow variables and log search for access are described in [25, 26, 51].

An alternate approach that was proposed for atomicity in a host equipped with persistent-memory is to use *aliasing* [15]. In this technique, accesses to transaction variables are redirected from their actual home memory to alternate aliased locations that are tracked using a software alias table. In [19] the aliasing was performed at the CXL controller, which aliased CXL transaction variables to alternate CXL locations, thereby allowing hosts to freely spill transaction variables and to ease durability implementation. The home CXL memory locations are only updated on a transaction commit.

In this paper we use aliasing to enforce atomicity within nodes that may not have HTM support. However, instead of a global alias table at the CXL controller that maps transactional CXL addresses to aliased CXL locations [19], we use *local aliasing* at each node. The alias table at a node maps global CXL transactional variables to local host memory addresses. Cache overflows simply update these host-local memory locations. A transaction abort discards the transaction's variables from the alias table, while a commit will initiate the update of the actual CXL memory locations. Note that alternatively, we can achieve in-host atomicity automatically using HTM (in hosts that support it) or with extensions to support unbounded-size transactions. More details are presented in Section 4.1.

Isolation can enforced using either conflict avoidance or detection and rollback. Conflict avoidance is usually implemented using locks; coarse-grained locking can serialize transaction executions but, although easy to implement, generally results in poor performance. Two-phase locking is a fine-grained locking scheme at the granularity of individual variables. However, to avoid deadlock, the locks need to be globally ordered, which restricts its use for ad-hoc transactions or requires complex deadlock detection and recovery schemes.

Optimistic concurrency control mechanisms allow transactions to execute concurrently without explicit locking, and rely on the transaction manager to detect potential access conflicts. HTM leverages the cache coherence mechanism on multi-core processors to implement conflict detection in hardware. The conflict detection mechanism is often simpler than the coherence protocol; for instance a write by a transaction to a cache line currently held in the read state by one or more other transactions can simply abort the writing thread; for coherence, the write must invalidate the copies in all the other threads and grant exclusive access to the writer. Software Transaction Memory (STM) uses software intervention of the memory accesses of a transaction, similar to a database transaction manager, to detect conflict and arbitrate transaction aborts.

In this paper we use a directory-based approach to maintain the transactions' access information and detect conflict. Using a software-based directory dovetails naturally with the alias table by simply maintaining additional status bits with the alias table entries and checking for conflict during table lookup. As in the case of atomicity enforcement, a hardware-based HTM , optionally extended for unbounded transactions, can be used to enforce isolation with the appropriate host hardware. More details are presented in Section 4.1.

3.2 Inter-node Transaction Management

The back-end controller must check a completed transaction for conflicts with transactions running on other nodes before allowing it to commit, and update CXL memory with the results of committed transactions.

Atomicity is easier to enforce at the back-end controller which is invoked only after a transaction has either aborted or completed at a node. Once a transaction variable has been accessed by any transaction running on a node, it remains in the node's memory hierarchy until it is explicitly written to the HTCXL controller on a transaction abort or commit. In the former case, the updates within the node are discarded, and HTCXL does not need to update CXL memory. If the transaction has completed, the HTCXL controller will check for potential conflicts with transactions on other nodes; if this transaction can safely commit, it writes all the updates of the transaction to CXL memory; else it asks the node to abort the transaction and erase all its buffered variables.

An additional consideration arises if the transaction requires full ACID semantics. In this case, the durability guarantees require that the updates of the transaction must be written atomically to non-volatile memory before allowing the transaction to commit. To guard against failure at the CXL subsystem, the updates must first be logged onto stable storage and then written to their persistent home locations. With a suitable recovery protocol in place, the transaction can be committed as soon as its updates are logged in stable storage, while the update of home locations occurs in the background. This step is not necessary for volatile transactions that typically only require ACI semantics.

Isolation at the CXL interface requires checking whether there is a conflict between a transaction on one node with a transaction on some other node. One of the guiding principles in our design is to

[1]An alternative approach updates the memory locations immediately but makes a copy of the current value to allow transaction rollback.

avoid frequent synchronous communication between the CXL controller and the nodes. Hence, we do not try to exploit the inter-node coherence mechanism proposed in the CXL 3.1 standard to detect concurrent access between nodes. There are currently no known implementations of this or performance studies that may suggest this is a viable approach.

For the HTCXL controller, we adopt a simplified directory-based scheme similar to the structure within a node. The controller keeps an access vector for each variable accessed by an active transaction. The access vector records the ids of the nodes that have checked out the variable. When a transaction aborts or commits, the HTCXL controller deletes the node from the access vectors of variables no longer cached in the node. These will be the variables written by the transaction (its *write set*) and the variables for which it was the only reader on that node (its *read set*). The proposed scheme is significantly simpler than the controller proposed in [19] using the global alias table with versioning information to detect conflict. More details are presented in Section 4.2.

4 OUR APPROACH

The two major components of our HTCXL design are shown in Figure 2. The first component is the *intra-host* transaction management system to serialize the transactions running on the host. We describe an implementation based on using the software *aliasing* technique [15, 19] to provide atomicity augmented with directory information to detect conflicts and enforce isolation. Alternatively, on an appropriate host, an HTM implementation could be used.

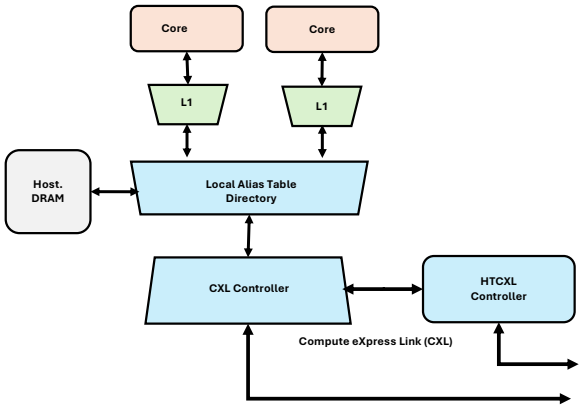


Figure 2: A high-level view of the placement of the HTCXL Controller within a system. Multiple hosts connect to the HTCXL Controller, which can either be embedded on the CXL Controller, operate as a stand alone CXL Device, or as a software only solution executing on a CXL Host.

The second component of our design is the *inter-host* back-end HTCXL Controller. The controller is concerned with detecting conflicts between transactions running on different nodes, aborting or committing transactions, and updating CXL memory with the results of committed transactions. If durability is desired, the controller must reliably persist the updates atomically on non-volatile storage despite unexpected failures. The HTCXL Controller can be implemented in a number of ways including embedded within a CXL Controller, implemented as a stand alone CXL device, or implemented as a software only solution executing on a CXL host.

The lifecycle of a thread is shown in Figure 3. Normally a core is executing non-transactional code of an application in the TxIDLE state. On executing a **BEGIN_Tx** instruction, the thread enters the TxEXECUTE state during which its reads and writes are arbitrated by the Local Alias Table manager. If a conflict with a transaction executing on another core in the same host is detected in the directory, one of the conflicting transactions must abort. If a transaction receives an NODE_ABORT (either a spontaneous self-abort or because it is the victim of a conflict), it returns to the non-transactional state from which it will retry the transaction after a random delay. When the transaction executes the **END_Tx** instruction, it transits to the TxEND state. At that time it interacts with the HTCXL controller to resolve any conflicts with transactions running on other nodes. The HTCXL controller will return a CXL_ABORT if it detects a conflict requiring this thread to abort; else it will return a CXL_COMMIT signal and the thread will successfully enter the TxCOMMIT state, and subsequently resume normal non-transactional execution.

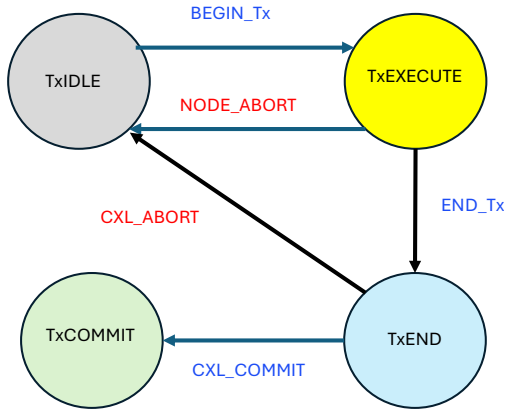


Figure 3: Transaction Lifecycle

4.1 Intra Node Transaction Management

In this section we present the design of the transaction management system within a node. It includes a local Alias Table that maps CXL addresses of transaction variables to node-local memory locations. An access vector is associated with each alias table entry creating a directory that tracks which cores have read or written the corresponding transaction variable. A software management system and library similar to that described in [15] converts transactional loads and stores to calls into the library, which provides the routines to manage the alias table.

Figure 4 shows the operation of the Alias Table with an example. Two transactions are running on cores 0 and 1 of some node. In Snapshot 1, core 0 makes a load request for transaction variable x; the library routine reads x from CXL memory and adds it to the local Alias Table implemented as a key value store. The access of x by the node is also noted by the HTCXL controller (see Section 4.2). A presence bit for core 0 is set to 1 in the Access Vector and the State field is set to to S (shared) indicating the variable is in a read state.

Core 0:	Core 1:
LOAD x;	STORE y;
	LOAD x;

Snapshot 1

Core 0:	Core 1:
LOAD x;	STORE y;
	LOAD x;
	STORE x;

Snapshot 2

Aliased Host Addresses	CXL Memory Address	Value	Access Vector	State
	&x	X_{V1}	[1 1 - -]	S
	&y	Y_{V1}	[- 1 - -]	M

Alias Table (Snapshot 1)

Aliased Host Addresses	CXL Memory Address	Value	Access Vector	State
	&x	X_{V1}	[1 ~~1~~ - -]	S
	&y	Y_{V1}	[- ~~1~~ - -]	~~M~~ I

Alias Table (Snapshot 2)

Figure 4: An example fragment of the Local Alias Table on a host with four cores. Two CXL variables x and y are aliased to the locations shown in the Alias Table. The Access Vector indicates which core(s) hold a copy of the variable. The STATE indicates whether the variable has been read (state S) or been written (state M). The Value field is the latest value of the variable on this node, which may be inconsistent with the values on other nodes.

The store access of y by core 1 reads y from CXL memory and installs it in the Alias Table in M state; the updated value of y is written to the Alias Table[2]. Core 1 then makes a load access for x whose value is read locally from the Alias Table and returned to core 0, *without* accessing CXL memory or informing the HTCXL controller. The access vector for x is updated to set the presence bit for core 1. The corresponding alias table following the operations of Snapshot 1 is shown in the first Alias Table of the figure.

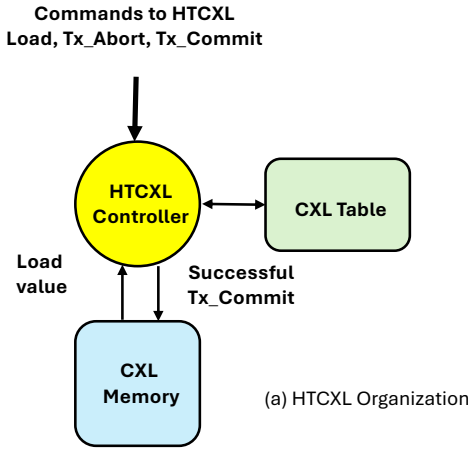
Snapshot 2 shows a continuation of execution where core 1 makes a store request for x. At this point the access vector indicates a conflict for x since core 0 has already read the older value of x, and one of the transactions must be aborted. In our protocol we abort the requester, core 1, while allowing core 0 to continue its execution without disturbance. This has the advantage of always aborting only one transaction even if multiple cores had copies of x at this time, and synchronously aborting it immediately. The figure shows that the Alias Table entry for y has been freed and the presence bit of core 1 for x has been deleted.

Handling Transaction Aborts: When a transaction aborts, its presence bit in the access vectors in the Alias Table must be reset; if no other presence bits are set, the entry can be reclaimed by setting its State to I. The HTCXL controller must be informed that these variables are no longer active at this node; the controller updates the state of the checked-out variables to avoid reporting false inter-node conflicts. A hardware-implemented directory can speed up this flash-reset operation in the node considerably.

Handling Transaction PreCommit: When a transaction executes **END_Tx** it signals its readiness to commit (pre-commit). However, it still needs to be checked for conflicts against transactions on other nodes. The node updates the access vectors just as in an abort operation, and creates a list of all variables that are no longer required in the node. The list is qualified by indicating whether the variables in the list are in the write or read set of the transaction. The list of addresses and the values of the variables in the write set are sent to the HTCXL controller, and the transaction waits for either a final *commit* or *abort* signal from the controller. An abort would cause it to retry the transaction after a random delay, otherwise it can continue with a fresh transaction.

4.2 Inter Node Transaction Management

In this section we describe the HTCXL controller that checks for inter-node transaction conflicts and updates home locations in CXL Memory with the final values of committed transactions.



CXL Memory Address	CheckOut Vector
&x	[1 1 0 ....0]
&y	[1 0 0 ....0]

(b) CXL Table

Node 0:
LOAD x;
Node 1:
STORE y;
LOAD x;

Figure 5: A high-level view of the operation of the HTCXL Controller with the transactions of Figure 4. A checkout bit is added to a CXLTable entry on a Load and deleted on a Tx_Abort or a Tx_Commit. The Read_Set and Write_Set are used to detect conflicts with the CXLTable state. Values are transferred to CXL Memory home locations if the transaction succeeds.

[2]Note the updates to the Alias Table occur asynchronously through normal processor cache writebacks.

Algorithm 1 HTCXL Controller Implementation

```
function HandleCommand(hostid, threadid)
    ABORT=FALSE;
    TxData = GetTxnInfo(hostid, threadid);
    case CXL Command:
  Load:
      Add hostid to CXL_Table[load_address]
      Read CXL Memory[load_address]
      Return memory value to hostid
  Tx_Abort:
  foreach address in UNCHECK
  Delete hostid from CXL_Table[address]
  Tx_Commit
  foreach address in WRITE_SET
      Check for conflict in CXL_Table[address]
  if conflict: ABORT = TRUE; break;
  foreach address in WRITE_SET
      Delete hostid from CXL_Table[address]
      if (ABORT== FALSE)
      Store data in CXL Memory[address]
  foreach address in READ_SET
      Delete hostid from CXL_Table[address]
      if (ABORT== TRUE) return TX_Abort
      else return TX_Commit
 end function
```

The HTCXL controller receives **Load**, **Tx_Abort** and **TX_Commit** commands from a node as shown in Figure 5. The controller maintains a table called the CXL Table that tracks the cache blocks that have been read from the CXL Memory; on receiving a Load command, the HTCXL controller adds the id of the requesting node to the Checkout Vector for that variable. When a transaction aborts, the node sends the controller a Tx_Abort command along with the list of variables that are no longer cached at the node; the HTCXL controller deletes the node id from the CXL Table entries for each of those addresses. On a TX_Commit, the controller receives a list of variables as a write set and a read set; HTCXL checks the Alias Table entry for each address in the write set to see if there is a conflict with another node. If the CXL Table indicates the variable has been checked out by another node then the transaction is marked and will be aborted. If none of the write set variables conflict, the transaction can safely commit. In either case, the node id must be removed from the checkout vectors for these variables in the CXL Table.

Note that the commit of multiple nodes can be checked simultaneously by concurrent threads at the HTCXL controller. While this may sometimes result in unnecessarily aborting a transaction due to unlucky timings, it should be noted that determining the optimal order to test for conflicts even in the off-line case is NP-complete [46], and any heuristic optimizations in our on-line and concurrent environment are unlikely to provide significant, if any, benefits.

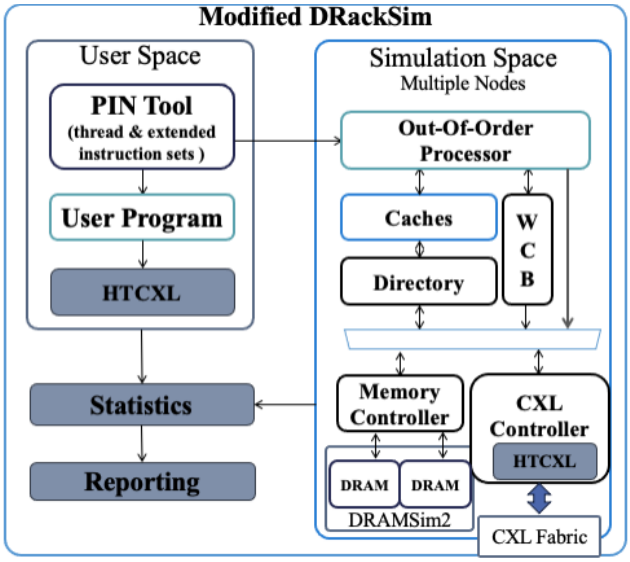


Figure 6: HTCXL simulation extends the DRackSim simulator. Applications are executed in user-space with HTCXL API, and PIN based traces are sent to Out-of-Order simulation with extensions for out HTCXL controller and memory backed by DRAMSim2.

5 EVALUATION

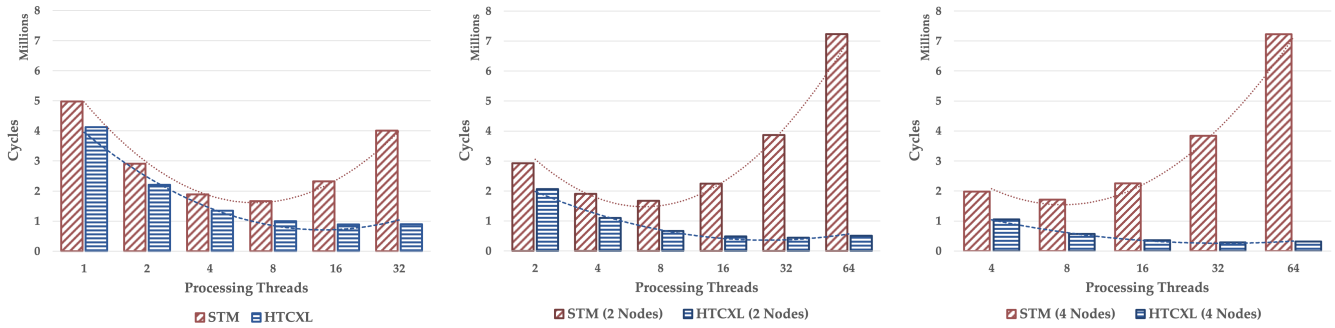
In the absence of readily available CXL 3.1 *Memory Sharing* systems with our embedded controllers, to evaluate our protocol and controller, we extend a cycle-accurate CXL simulator DRackSim [41] to model our controller and communicate with our library. We create an HTCXL library that is linked to benchmark applications. We compare our approach using both microbenchmarks and benchmarks from the Stanford Transactional Applications for Multi-Processor (STAMP) benchmark suite [34]. We utilize the Chameleon cluster infrastructure [28] for executing simulations.

5.1 Experimental Setup

DRackSim models the out-of-order processor micro-architecture at the cycle level and includes a detailed initialization and timing configuration, including CXL fabric topology. DRackSim utilizes DRAMSim2 [44] for cycle-accurate memory simulation and Pin, a dynamic binary instrumentation tool [33], to decouple execution from simulation. Instruction traces are execution-driven and piped to the simulator in real-time. Traces may also be saved for offline analysis. The simulator, along with our modifications, is outlined in Figure 6.

The rate of incoming transactions is determined, in effect, by the rate at which the PIN instrumented program runs. To synchronize timing in the decoupled environment, we capture timing events in the instrumented program evaluated and align the events in the simulation space of the HTCXL simulated hardware. We made several extensions to the DRackSim based setup:

- Added PIN support for streaming (non-temporal) stores and persistent memory flushes.



(a) 1 Node **(b) 2 Nodes** **(c) 4 Nodes**

Figure 7: Micro-benchmark of transactional updates into a CXL based hash table using concurrent processing threads on a CXL Fabric of 4 Nodes. Each transaction is 50 writes to random locations.

- Implemented statistics gathering routines for operation counting.
- Added write-combining and internal buffering for pending write requests in the controllers.
- Added instrumentation for determining bounding memory regions for local DRAMand remote CXL regions to pass to corresponding evaluations.
- Extended the simulator to support our HTCXL controller operations, checking out memory reads, capturing write-sets, and notifying a node of transaction completion.
- Added support to map multiple simulation execution traces from threads in a single application across a configurable number of simulated nodes. This allows a single multi-threaded benchmark to easily share a concurrent address space across the simulated CXL fabric.

We build each benchmark using a configuration for our approach and a configuration for Software Transaction Memory, STM. To evaluate performance against an STM, we utilize the TL2 or Transactional Locking approach [10]. The approach utilizes software to capture loads and stores, and forward stores to future loads. Loads are held until the transaction commits, at which point the software validates that transactional variables do not conflict before copying the successful transaction write set to main memory. We augment the TL2 library to utilize the begin and end simulation library calls, along with notifications for memory allocations. Memory allocations for transactional bookkeeping internal to a transaction are made to local memory to ensure the comparison does not create extra CXL traffic.

In our HTCXL implementation, the Hierarchical Transactional CXL controller handles multiple nodes attached. In this configuration, multiple nodes, each with its own concurrent transactions on the local node, are connected via CXL to our HTCXL controller. For our evaluation, we create a local HTCXL library for intra-node conflict detection. When performing a transactional write, the write is forwarded to our controller asynchronously to avoid bulk sending on transaction end. The controller validates the read and write sets as described above, utilizing pipelined comparison buffers that check the latest versions of the read set and acquire locks on the write set. Once the transaction is validated, the node is notified, and the core can continue execution of the thread.

For our evaluation, we set the number of nodes to 4 and the number of cores for each node to 32. We extend the interleaving of thread traces across the nodes, and simulate up to 64 threads spread evenly across the nodes. Each node has enough cores to schedule 100% execution of each processing thread on the hardware. With HTCXL across multiple nodes, we send the read and write sets to the controller.

5.2 Micro-benchmarks

In our micro-benchmarks, we create a simple *hash−table* in CXL memory spread across the nodes and perform a series of transactions on the table. Each transaction performs a configurable number, by default 50, random updates into the table. The number of concurrent thread workers is configurable along with the number of reads and transaction size. This workload stresses the underlying memory system and implementation of transactional guarantees without performing much computation work itself.

We first vary the number of processing threads for a transactional update of 50 elements into the table. Figure 7 shows the benchmark total time in cycles to perform 256 transactional updates into the CXL hash table. As shown in the figure 7a for one node, all conflict detection is performed locally, and our HTCXL controller is not used for conflict detection between nodes. Since the simulated out-of-order processor is configured for 32 cores, the simulation for one node only is up to 32. For a single node, we scale slightly better than STM due to using block-based locations for read and write sets instead of linked lists. Additionally, we perform conflict detection in the alias table directly and eagerly at and within the node, which is beneficial for this workload.

In figure 7b, when using two nodes, both the STM and our method HTCXL perform better. However, HTCXL realizes a larger decrease in application performance time with the controller. Similarly, for four nodes in figure 7c, there is a slight increase in STM for communication across the fabric while HTCXL improves through our protocol. This improvement is highlighted in Figure 8, where we show that, with 16 and 32 threads, the decrease in application time is greater using our approach.

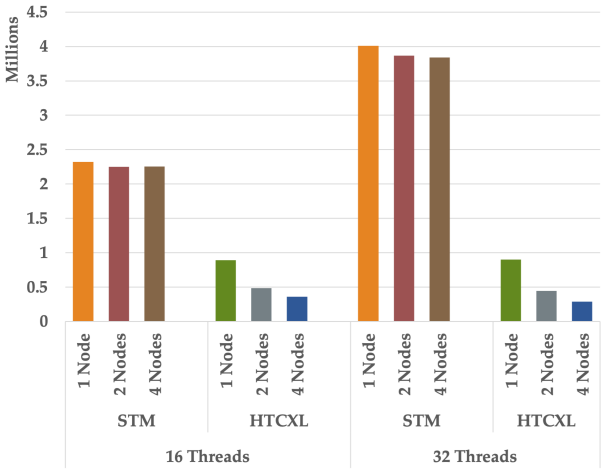


Figure 8: Execution time for micro-benchmark using 16 and 32 threads with 50 elements across a varying number of nodes in the CXL fabric.

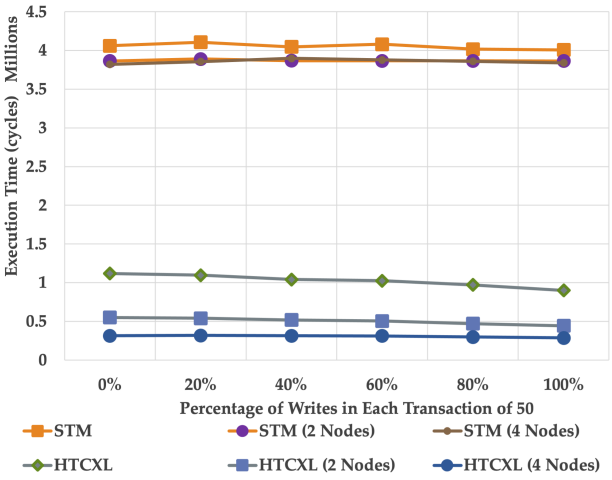


Figure 10: Execution time for micro-benchmark with a varying number of writes in a 50-element transaction and 32 threads.

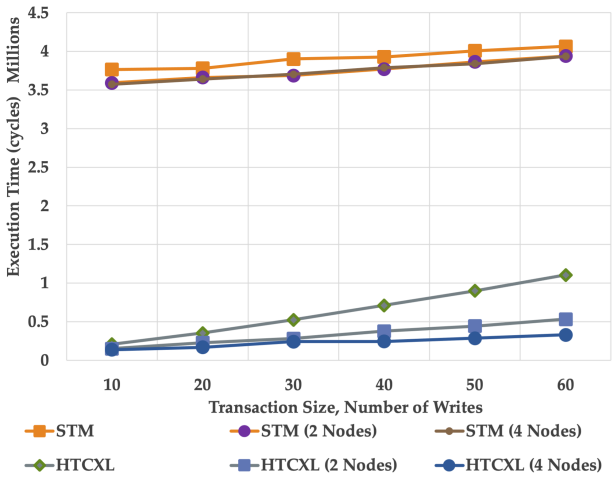


Figure 9: Execution time for micro-benchmark *hash-table* with 32 threads performing a varying number of transactional writes in each update.

Next, we examined the effect of transaction size on the execution time of the workload. Figure 9 shows the effect of increasing the number of writes in a transaction. In this setup, we spawn 32 threads, and each thread performs the configured number of writes in each transaction. We analyze from 10 to 60 writes in a transaction and record the benchmark execution times. On a single node, our approach is more affected by the number of writes when compared to STM . In this case, we are performing eager aborting and conflict detection, and in the absence of reads, an abort is not needed. As more nodes are added, our HTCXL approach is less affected by the size of the transaction.

Finally, in figure 10, we show increasing the percentage of writes in a 50-element transaction. In both the STM and our HTCXL approaches, using software, a read to a memory location in a transaction must be checked against previous writes in the transaction. Our approach is more affected by reads as we must read a value from CXL memory into the local node alias table while also notifying our HTCXL controller that the variable has been checked out for reading.

5.3 Benchmarks

In this section, we evaluate our approach using two benchmarks from the STAMP benchmark suite [34]. For each benchmark, we examine both a *low* and *high* contention for shared data among concurrent threads. In each configuration, we utilize four nodes in the CXL fabric with concurrent threads spread evenly across the nodes.

We first examine *kmeans*, the machine learning K-means algorithm benchmark from MineBench [36], where data is partitioned into K clusters from objects in an N-dimensional space. In this benchmark, as the means are calculated and cluster centers updated over each partition, the critical sections are protected by transactions, where the size of the transaction is proportional to the N-dimensional space. Transactions in the *kmeans* data set are generally categorized with small read/write set lengths and short transaction times with lower contention.

In the first configuration of *kmeans*, *kmeans-low*, the number of clusters is configured to 40 with 512 data points, eight dimensions, and eight centers. Figure 11 shows the application execution time for the number of processors. As shown in the figure, due to the lower contention, both STM and HTCXL methods scale well until about 16 processes. At this point, STM starts to incur more overhead and communication among the nodes, and its scalability stops, while HTCXL continues to scale to 32 processing threads. At 64 threads, HTCXL outperforms STM by a factor of almost nine. Figure 12 shows the number of aborts for HTCXL broken down at the controller and

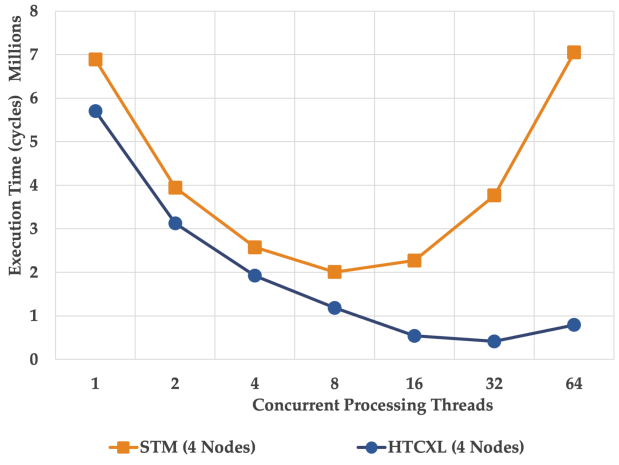


Figure 11: Execution time for calculating 40 clusters in the application *kmeans-low* benchmark using mutliple processes on one or more nodes on CXL Fabric of 4 Nodes.

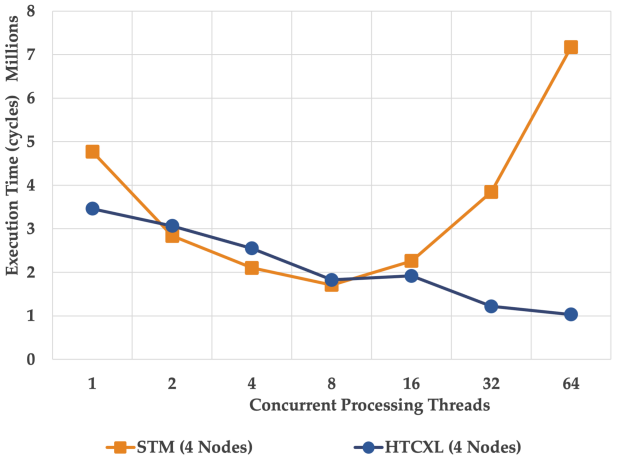


Figure 13: Execution time for calculating 15 clusters in the application *kmeans-high* benchmark using mutliple processes on one or more nodes on CXL Fabric of 4 Nodes.

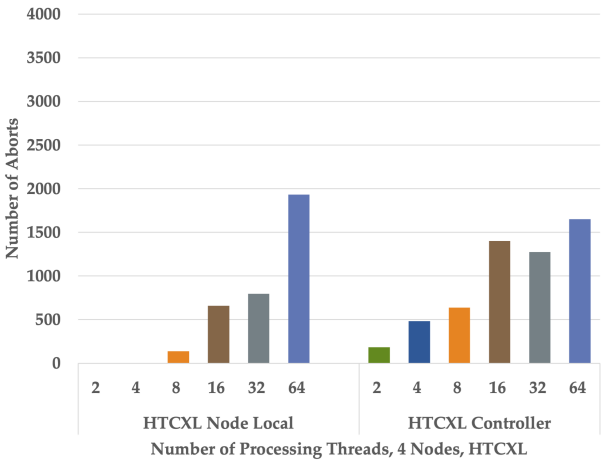


Figure 12: The number of aborts from the node and controller with HTCXL method for the application *kmeans-low* benchmark on CXL Fabric of 4 Nodes.

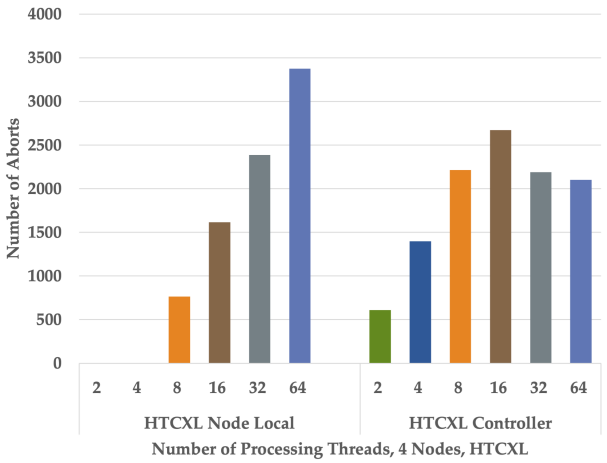


Figure 14: The number of aborts from the node and controller with HTCXL method for the application *kmeans-high* benchmark on CXL Fabric of 4 Nodes.

local node-initiated aborts. The figure illustrates the critical time required to transition from 32 to 64 concurrent processes, or 8 to 16 threads at each node. The controller begins initiating more inter-node aborts when the system processes increase from 8 to 16, corresponding to the diminishing returns in lower execution times for HTCXLwith increasing processing threads. Additionally, the local node initiates more aborts when the number of concurrent threads on a node increases from 8 to 16 (corresponding to an increase from 32 to 64 processing threads in the system).

The *kmeans-high* configuration is depicted in the next set of figures. In this configuration, the same number and dimensionality of input data points are used as in the prior experiment; however, the number of clusters is reduced from 40 to 15, which increases contention and decreases overall execution time. With the increased contention from concurrent calculating processing threads, the scalability of the benchmark is reduced, and more aborts are incurred. Figure 13 shows the execution time for both STM and HTCXL for an increasing number of threads. The execution time for calculating only 15 clusters, as compared to 40 in the previous experiment, is reduced. While the execution time is reduced, the scalability of adding additional processing threads is also reduced due to increased contention. This slower performance is due to an increase in contention and aborts, as shown in Figure 14. STM outperforms HTCXL for a select number of processes due to the high number of aborts. When compared to *kmeans-low*, there is a 50% increase in aborts during transactional updates with 64 concurrent processors,

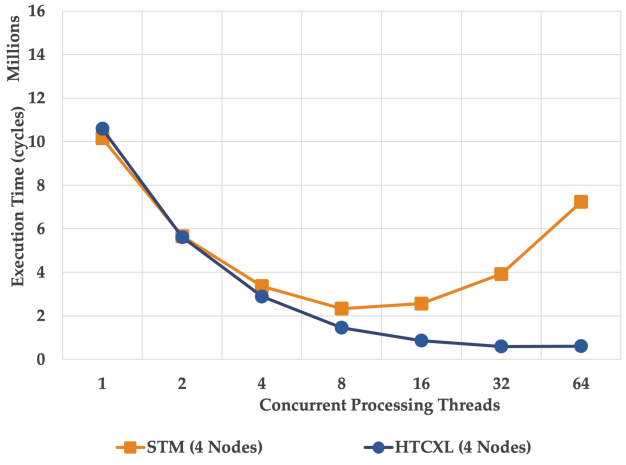


Figure 15: Execution time for the application *vacation-low* benchmark using mutliple processes on one or more nodes on CXL Fabric.

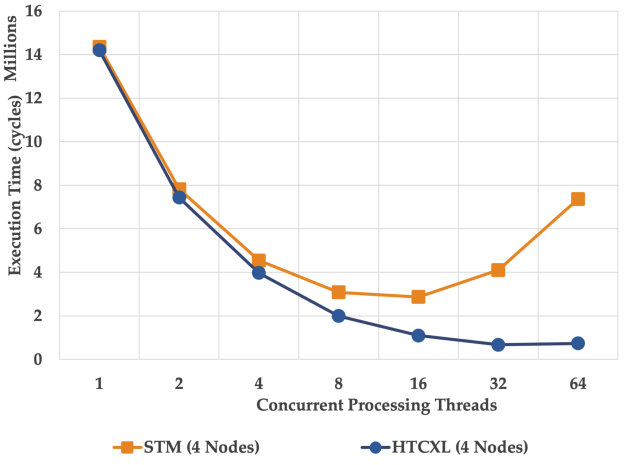


Figure 17: Execution time for the application *vacation-high* benchmark using mutliple processes on one or more nodes on CXL Fabric.

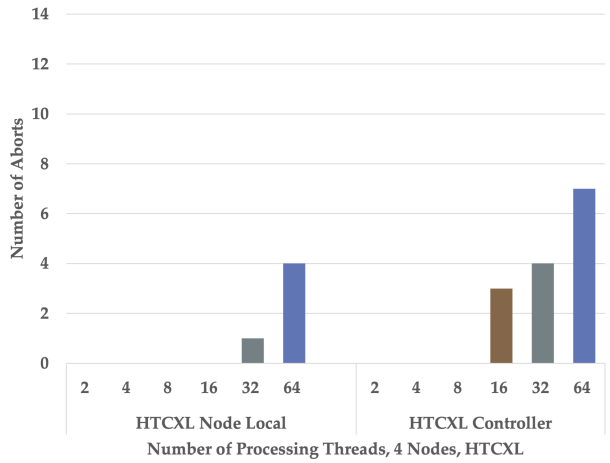


Figure 16: The number of aborts from the node and controller with HTCXL method for the application *vacation-low* benchmark on CXL Fabric of 4 Nodes.

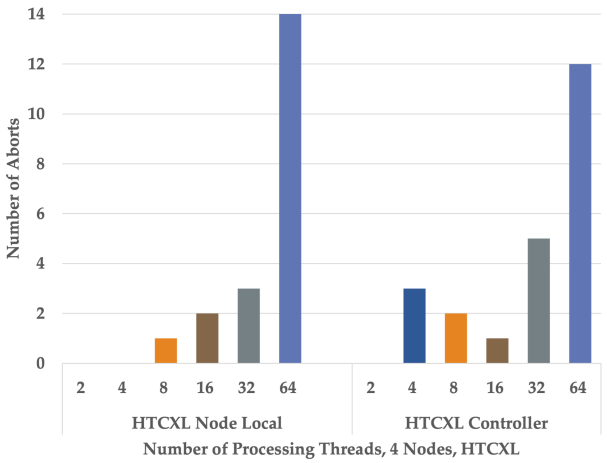


Figure 18: The number of aborts from the node and controller with HTCXL method for the application *vacation-high* benchmark on CXL Fabric of 4 Nodes.

and over double the number of aborts for a lower number of concurrent processors. At 64 concurrent processing threads across the CXL fabric, our approach HTCXL still outperforms STM by a factor of seven.

Our final benchmark evaluation is *vacation*, which implements an online travel reservation transaction processing system. It utilizes a set of trees to store customer information, and threads interact with this information transactionally to maintain the integrity of the data. The characteristics of the workload include medium-length transactions and read/write sets with long execution times.

In the first configuration, *vacation-low*, the database maintains 16k records with concurrent update threads. Each transaction updates at most 2 records over 90% of the database. The execution time for increasing numbers of processors for the same number of updates is shown in Figure 15. Both STM and our HTCXL methods scale well to 8 concurrent threads, with data processing in separate areas of the tree. After eight threads, our approach continues to scale well across the four nodes and conflict detection is split between the nodes and the controller. The number of aborts initiated by nodes and our controller is shown in Figure 16. Due to the low contention, only a few aborts are initiated in either area, with the controller initiating only six aborts for 64 concurrent processing threads. At 64 processing threads, our approach outperforms STM by a factor of over ten.

The following configuration is *vacation-high*, which simulates the same database but with a different transactional profile. In the

vacation-high profile, contention increases, with sessions reducing to a smaller fraction of the database (60%) to concentrate operations while also doubling the number of operations in each transactional session. Figure 17 shows the execution time for *vacation-high* for both STM and HTCXL. In higher contention with increased conflicts among concurrent updates, both approaches are much slower, with nearly a 40% increase in benchmark execution time. The number of aborts is shown in Figure 18, where at 64 concurrent processing threads the number of aborts jumps by over a factor of four at the node and over doubles at the controller. With the increase in contention and transaction aborts the benchmark scalability flattens for HTCXL. Still, at 64 processing threads and four nodes, our approach outperforms a pure STM approach on the CXL fabric by a factor of ten.

6 RELATED WORK

The hardware technologies Compute eXpress Link (CXL), Persistent Memory (PM), and Hardware Transactional Memory (HTM) are discussed in Section 2 above. The combination of these technologies to create ACID support for Memory Transactions in a disaggregated memory architecture are reviewed in detail throughout this section.

Related CXL Work: CXL [8] can have significant benefits in scale-up architectures for database engines and systems [30]. Pond is a CXL memory pooling systems for cloud platforms [31]. CXL over Ethernet was explored in [49] using FPGA, data caching and congestion controls. DirectCXL [20] connects a host processor with remote memory resources enabling loads and stores to remote memory resources. However, the above works do not address Persistent Memory, much less any ACID requirements for transactions in CXL-based memory servers.

Recent work such as [52] creates a distributed memory management system based on reference counting. However, these approaches do not address ACID requirements for transactions in CXL-based memory servers.

Related Persistence Work: Analysis of consistency models for persistent memory was considered in [39]. Changes to the front-end cache for ordering cache evictions were proposed in [7, 27, 47, 53]. BPFS [7] proposed *epoch barriers* to control eviction order, while [47] proposed a *flush* software primitive to control of update order. Snapshotting the entire micro architectural state at the point of a failure is proposed in [35]. A non-volatile victim cache to provide transactional buffering was proposed in [53], with the added property of not requiring logging, but requires changes to the front-end cache controller to track pre- and post- transactional states for cache lines in both volatile and persistent caches, atomically moving them to durable state on transaction commits.

Memory controller support for transaction atomicity in persistent memory have been proposed in [11, 14, 16, 40, 42, 54, 56]. Adding a small DRAM buffer in front of persistent memory to improve latency and to coalesce writes was proposed in [42]. The use of a volatile victim cache to prevent uncontrolled cache evictions from reaching PM was described in [11, 14, 40], but requires software locking for concurrency control. FIRM [54] describes techniques to differentiate persistent and non-persistent memory traffic, and presents scheduling algorithms to maximize system throughput and fairness. Low-level memory scheduling to improve efficiency of persistent memory access was studied in [56]. Except for [11, 14, 40], none of these works deal with the issues of atomicity or durability of write sequences. Our approach effectively uses HTM for concurrency control and does not require changes to the font-end cache controller or use logs for replaying transactions to PM.

Related Concurrency Work: Existing non-HTM solutions, such as Mnemosyne [48], ATLAS [5], and REWIND [6], tightly couple concurrency control with durable writes of either write-ahead logs or data updates into persistent memory to maintain persistence consistency. Software that employs these approaches generally means they must extend the duration for which they remain in critical sections, leading to longer times to hold locks, which reduces concurrency and expands transactional duration.

Other concurrency related work [15, 32] decouples concurrency control so that post transactional values may flow through cache hierarchy and reach PM asynchronously; however, the write ahead log for an updating transaction has to get committed into PM synchronously before the transaction can close so that the integrity of the foreground value flow is preserved across machine restarts. Another hardware-assisted mechanism proposes hardware changes to allow a dual-scheme checkpointing that writes previous checkpointed values in the background while collecting current transaction writes [43].

Related HTM Work: The capacity of HTM transactions are increased by introducing a software layer for version implementation using Snapshot Isolation [12]. Some work addresses improving conflict management for HTM such as LosaTM [13], which provides a low-overhead conflict manager, and other hardware based strategies such as PleaseTM [38], a requestor-wins strategy [9], and ForgiveTM [37]. LogTM-SE [51] proposes decoupling HTM from caches using an undo log and signatures, allowing for an update to memory in-place and unbounded nesting, context switching, and other migrations and allows values to spill from the L1 all the way to memory, since the original value of an aborted transaction can be restored from the undo log. However, these works do not address persistence and durability of transactions onto a non-volatile media such as persistent memory.

Related Persistence + HTM Work: Some work [4, 17, 18] utilizes un-modified HTM for concurrency control decoupled from persistence to HTM. cc-HTM [17] introduces the concept of adjustable *lag* whereby users can allow transaction execution to continue in fast cache with selectable PM durability guarantees on the back-end. NV-HTM [4] must wait for prior transactions to complete before making forward progress. Hardware Transactional Persistent Memory, or HTPM [18], utilizes HTM for concurrency control and isolation, with a back-end memory controller based on [11, 40]. While HTPM requires no changes to current HTM semantics or additions to the cache policies, it is bound to a single host and HTM limits.

Other work [2, 3, 26, 32, 50] requires making significant changes to the existing HTM semantics and implementations. For instance, PHTM [3] and PHyTM [2], propose a new instruction called *TransparentFlush* which can be used to flush a cache line from *within a transaction* to persistent memory without causing any transaction to abort. Similarly, for DUDETM [32] to use HTM, it requires that designated memory variables *within a transaction* be allowed to be updated globally and concurrently without causing an abort. Durable HTM (DHTM) [26], changes the coherence protocol through

hardware changes, and using re-do logging provides for durability of PM transactions. DHTM is size limited to the LLC and log writes bypass the LLC. Logging based software approaches are problematic for HTM transactions (e.g., Intel TSX) which cannot bypass the caches in order to flush the log records synchronously into persistent memory ahead of transaction closings. To log within a transaction, PTM [50] proposes changes to processor caches while adding an on-chip scoreboard and global transaction id register to couple HTM with PM. Unbounded HTM (UTHM) [25] provides unbounded transactions using address signatures for overflowed blocks and hybrid logging with an undo log for DRAM and a redo log for NVM.

7 SUMMARY

To support high-performance applications across cutting-edge domains, modern servers need to handle concurrent applications on shared data sets. To support these applications, disaggregated memory servers using CXL are becoming increasingly popular. However, standard mechanisms to handle memory transactions on multi-node CXL systems face many challenges.

In this paper, we introduced HTCXL, which provides ACID support for memory transactions in a CXL-based memory architecture. A hierarchical approach where transaction mechanisms at the nodes and the CXL controller work synergistically, with each domain using the most suitable mechanism. Atomicity, Isolation, and Durability mechanisms are decoupled, allowing the best approach to be used at different entities.

We evaluated HTCXL using a cycle-accurate simulator and multiple microbenchmarks and transactional benchmarks from the STAMP suite. We demonstrated that HTCXL continues to scale well across multiple nodes in the CXL fabric, achieving significant improvements over an STM approach.

REFERENCES

[1] ARM. 2022. Overview of Arm Transactional memory Extension. https://developer.arm.com/documentation/102873/0100/Hardware-Transactional-Memory

[2] Hillel Avni and Trevor Brown. 2016. PHyTM: Persistent Hybrid Transactional Memory. *Proceedings of the VLDB Endowment* 10, 4 (2016), 409–420.

[3] Hillel Avni, Eliezer Levy, and Avi Mendelson. 2015. Hardware Transactions in Nonvolatile Memory. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363* (Tokyo, Japan) *(DISC 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 617–630.

[4] Daniel Castro, Paolo Romano, and João Barreto. 2018. Hardware Transactional Memory Meets Memory Persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 368–377. https://doi.org/10.1109/IPDPS.2018.00046

[5] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[6] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment* 8, 5 (2015), 497–508.

[7] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[8] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* (jun 2024). https://doi.org/10.1145/3669900 Just Accepted.

[9] Dave Dice, Maurice Herlihy, and Alex Kogan. 2018. Improving Parallelism in Hardware Transactional Memory. *ACM Trans. Archit. Code Optim.* 15, 1, Article 9 (mar 2018), 24 pages. https://doi.org/10.1145/3177962

[10] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Distributed Computing*. Springer, 194–208.

[11] Kshitij A. Doshi, Ellis R. Giles, and Peter J. Varman. 2016. Atomic Persistence for SCM with a Non-intrusive Backend Controller. In *The 22nd International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 77–89. https://doi.org/10.1109/HPCA.2016.7446055

[12] Ricardo Filipe, Shady Issa, Paolo Romano, and João Barreto. 2019. Stretching the capacity of hardware transactional memory in IBM POWER architectures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM. https://doi.org/10.1145/3293883.3295714

[13] Chao Fu, Li Wan, and Jun Han. 2022. LosaTM: A Hardware Transactional Memory Integrated With a Low-Overhead Scenario-Awareness Conflict Manager. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 4849–4862.

[14] Ellis Giles, Kshitij Doshi, and Peter Varman. 2013. Bridging the Programming Gap Between Persistent and Volatile Memory Using WrAP. In *Proceedings of the ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '13)*. ACM, New York, NY, USA, Article 30, 10 pages. https://doi.org/10.1145/2482767.2482806

[15] E.R. Giles, K. Doshi, and P. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. 1–14. https://doi.org/10.1109/MSST.2015.7208276

[16] Ellis Giles, Kshitij Doshi, and Peter Varman. 2017. Brief Announcement: Hardware Transactional Storage Class Memory. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (Washington, DC, USA) *(SPAA '17)*. ACM, New York, NY, USA, 375–378. https://doi.org/10.1145/3087556.3087589

[17] Ellis Giles, Kshitij Doshi, and Peter Varman. 2017. Continuous Checkpointing of HTM Transactions in NVM. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management* (Barcelona, Spain) *(ISMM 2017)*. ACM, New York, NY, USA, 70–81. https://doi.org/10.1145/3092255.3092270

[18] Ellis Giles, Kshitij Doshi, and Peter Varman. 2018. Hardware Transactional Persistent Memory. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, Virginia, USA) *(MEMSYS '18)*. Association for Computing Machinery, New York, NY, USA, 190–205. https://doi.org/10.1145/3240302.3240305

[19] Ellis Giles and Peter Varman. 2025. ACID Support for Compute eXpress Link Memory Transactions. In *Proceedings of the SC '24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis* (Atlanta, GA, USA) *(SC-W '24)*. IEEE Press, 982–995. https://doi.org/10.1109/SCW63240.2024.00138

[20] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 287–294. https://www.usenix.org/conference/atc22/presentation/gouk

[21] Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory, 2nd Edition* (2nd ed.). Morgan and Claypool Publishers.

[22] Intel Corporation. 2012. Intel Transactional Synchronization Extensions. In *Intel Architecture Instruction Set Extensions Programming Reference*. Chapter 8. http://software.intel.com/.

[23] Intel Corporation. 2014. Intel Architecture Instruction Set Extensions Programming Reference. http://software.intel.com/.

[24] Intel Corporation. 2022. "Intel 64 and IA-32 Architectures Software Developer Manual". http://www.software.intel.com/.

[25] Jungi Jeong, Jaewan Hong, Seungryoul Maeng, Changhee Jung, and Youngjin Kwon. 2020. Unbounded hardware transactional memory for a hybrid DRAM/NVM memory system. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 525–538.

[26] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. Dhtm: Durable hardware transactional memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 452–465.

[27] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[28] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[29] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. 2015. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development* 59, 1 (2015), 8:1–8:14. https://doi.org/10.1147/JRD.2014.2380199

[30] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. arXiv:2401.01150

[31] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 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, Volume 2*. 574–587.

[32] 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 Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, Vol. 40. ACM, 190–200.

[34] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 35–46.

[35] D. Narayanan and O. Hodson. 2012. Whole-System Persistence. In *Proceedings of 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (London, UK). ACM Press, 401–410.

[36] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. 2006. Minebench: A benchmark suite for data mining workloads. In *2006 IEEE International Symposium on Workload Characterization*. IEEE, 182–188.

[37] Sunjae Park, Christopher J Hughes, and Milos Prvulovic. 2019. ForgiveTM: Supporting lazy conflict detection in eager hardware transactional memory. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 192–204.

[38] Sunjae Park, Milos Prvulovic, and Christopher J Hughes. 2016. PleaseTM: Enabling transaction conflict management in requester-wins hardware transactional memory. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 285–296.

[39] Steven Pelley, Peter M Chen, and Thomas F Wenisch. 2014. Memory persistency. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 265–276.

[40] Libei Pu, Kshitij A. Doshi, Ellis R. Giles, and Peter J. Varman. 2016. Non-Intrusive Persistence with a Backend NVM Controller. *IEEE Computer Architecture Letters* 15, 1 (Jan 2016), 29–32. https://doi.org/10.1109/LCA.2015.2443105

[41] Amit Puri, Kartheek Bellamkonda, Kailash Narreddy, John Jose, Venkatesh Tamarapalli, and Vijaykrishnan Narayanan. 2024. DRackSim: Simulating CXL-enabled Large-Scale Disaggregated Memory Systems. In *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (Atlanta, GA, USA) *(SIGSIM-PADS '24)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/3615979.3656059

[42] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. ACM, New York, NY, USA, 24–33. https://doi.org/10.1145/1555754.1555760

[43] 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*. ACM, 672–685.

[44] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters* 10, 1 (2011), 16–19. https://doi.org/10.1109/L-CA.2011.4

[45] Das Sharma and Ishwar Agarwal. 2024. Compute Express Link 3.0 White Paper. https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_3.0_white-paper_FINAL.pdf

[46] Gokarna Sharma. 2014. *Scheduling in Transactional Memory Systems: Models, Algorithms, and Evaluations*. Louisiana State University and Agricultural & Mechanical College.

[47] S. Venkatraman, N. Tolia, P. Ranganathan, and R. H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte Addressable Memory. In *Proceedings of 9th Usenix Conference on File and Storage Technologies* (San Jose, CA, USA). ACM Press, 61–76.

[48] H. Volos, A. J. Tack, and M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, CA, United States). ACM Press, 91–104.

[49] Chenjiu Wang, Ke He, Ruiqi Fan, Xiaonan Wang, Wei Wang, and Qinfen Hao. 2023. CXL over Ethernet: A Novel FPGA-based Memory Disaggregation Design in Data Centers. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing*

Machines (FCCM). 75–82. https://doi.org/10.1109/FCCM57271.2023.00017

[50] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen. 2015. Persistent Transactional Memory. *IEEE Computer Architecture Letters* 14, 1 (Jan 2015), 58–61. https://doi.org/10.1109/LCA.2014.2329832

[51] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 261–272. https://doi.org/10.1109/HPCA.2007.346204

[52] 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* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 658–674. https://doi.org/10.1145/3600006.3613135

[53] 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* (Davis, California) *(MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744

[54] Jishen Zhao, Onur Mutlu, and Yuan Xie. 2014. FIRM: Fair and high-performance memory control for persistent memory systems. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 153–165.

[55] Christopher Zhou, Ellis Giles, and Peter Varman. 2025. Transaction Architecture for CXL Memory. In *Proceedings of the 2024 9th International Conference on Cloud Computing and Internet of Things (CCIOT '24)*. Association for Computing Machinery, New York, NY, USA, 83–91. https://doi.org/10.1145/3704304.3704316

[56] Ping Zhou, Yu Du, Youtao Zhang, and Jun Yang. 2010. Fine-grained QoS scheduling for PCM-based main memory systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.