DIKTAMO: Extending CXL for Resilience to CPU Failures

Antonis Psistakis, Burak Ocalan, Chloe Alverti[§], Fabien Chaix[◇], Ramnatthan Alagappan, and Josep Torrellas
University of Illinois Urbana-Champaign, USA [§]National Technical University of Athens, Greece [◇]FORTH, Greece
{psistaki, bocalan, ramn, torrella}@illinois.edu, xalverti@cslab.ece.ntua.gr, chaix@ics.forth.gr

***Abstract*—Compute Express Link (CXL) 3.0 and beyond allows the compute nodes of a cluster to share data with hardware cache coherence and at the granularity of a cache line. This enables shared-memory semantics for distributed computing, but introduces a new resilience challenge: a node failure leads to the loss of the dirty data in its caches, corrupting application state. Sadly, the CXL specification does not consider processor failures. Moreover, when a component fails, the specification tries to isolate it and continue application execution; there is no attempt to bring the application to a consistent state—a step required to recover a shared-memory program.**

To address these limitations, this paper extends CXL to be resilient to node failures, and to correctly recover the application after node failures. We call the system *DIKTAMO*. To survive node failures, DIKTAMO augments the coherence transaction of a write with messages that propagate the update to a small set of other nodes (i.e., *Replicas*). Replicas save the update in a local hardware Logging Unit. Replication ensures resilience to node failures. Then, at regular intervals, the Logging Units dump the compressed updates to memory. After a node failure, recovery involves using the logs to bring the directory and memory to a correct state. Our evaluation with 16 4-core nodes shows that DIKTAMO enables fault-tolerant execution with 30% slowdown with 3 replicas (or 27% with 2 replicas) over a platform without fault-tolerance support. DIKTAMO is 2.82x faster than ensuring fault tolerance by using a write-through protocol.

***Index Terms*—Compute Express Link, CXL, distributed shared memory, cache coherence, fault tolerance.**

I. Introduction

The latest versions of the Compute Express Link (CXL) standard [10]–[12], including CXL 3.0 and beyond, introduce hardware support for distributed compute nodes (CNs) to access shared-data stored in Memory Nodes (MNs) with hardware-enforced cache coherence at the granularity of a cache line. This capability blurs the line between shared-memory multiprocessors and distributed clusters, raising questions about how distributed applications can benefit from shared-memory programming semantics [20], [27], [28], [39], [45], [47]. However, this same capability complicates a common challenge in distributed computing: tolerance to faults. Traditionally, distributed applications such as databases and key-value stores share data at coarse granularities (e.g., records) and handle faults with relatively high overhead software techniques such as data replication or logging. With CXL, a different approach is needed, as data sharing occurs transparently in hardware and at cache-line granularity.

The CXL specification describes support for reliability, availability, and serviceability (RAS) [10], [17], [18]. While the support is extensive, it has two limitations. First, it does not consider processor (i.e., host) failures; it only considers link, switch, and device (e.g., memory) failures. The second limitation is that, when a component fails, the specification tries to isolate it. There is no attempt to recover the workload to a consistent state.

In reality, surviving processor failures is crucial, as CPUs are responsible for many failures in datacenters [4]. Moreover, when a node participating in a shared-memory program fails, the dirty data in its caches is lost. Hence, simply isolating the node and continuing execution results in using inconsistent application state. Correct execution requires first the recovery of the application state.

To address these limitations, this paper proposes an extension to CXL to: 1) be resilient to the failure of CNs and 2) correctly recover the application after CN failure. We call the system *DIKTAMO*. DIKTAMO's extensions are mostly focused on the CXL Transaction Layer, and minimally affect the CXL Link Layer.

To handle the failure of CNs, DIKTAMO augments the coherence transaction of a write with messages that propagate the update to a small set of other CNs (i.e., the *Replicas*). The replicas save the update in a local hardware *Logging Unit*. Such replication ensures resilience to CN failures. The Logging Units compress the data in their logs and, at regular intervals, dump it to the memory of Memory Nodes (MNs), which is resilient.

CN failure is detected by minimally extending the CXL Link Layer. Recovery is a distributed software-driven process that involves reading the directory state, reading the logs in the Logging Units, and updating the directory and memory state. The recovery restores a consistent application state, and enables forward progress of the application on the healthy nodes.

We evaluate DIKTAMO using simulations of a cluster of 16 CNs of 4 cores each, and 16 MNs interconnected via a CXL switch. We run parallel engineering applications and the YCSB key-value store. We find that DIKTAMO enables fault-tolerant execution with 30% slowdown with three replicas (or 27% with two replicas) over the same platform without fault-tolerance support. Further, DIKTAMO is 2.82x faster than ensuring fault tolerance by using write-through writes to MNs. We also evaluate the impact of various aspects of DIKTAMO.

The contributions of this work are:

2379-3155/26/$31.00 ©2026 IEEE
DOI 10.1109/MICRO67795.2026.00155

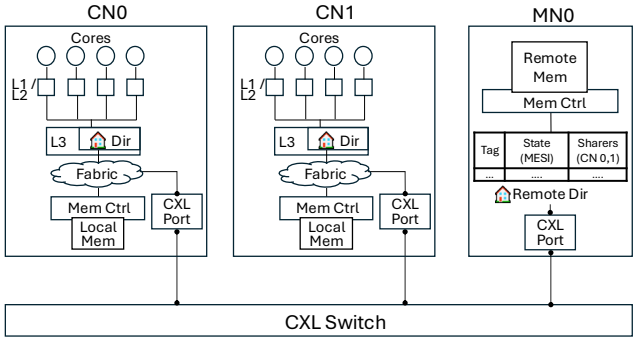


Fig. 1: CXL-enabled distributed shared-memory system.

- The DIKTAMO architecture that extends CXL to make it resilient to processor failures.
- The DIKTAMO cache coherence protocol, which extends a traditional MESI protocol with data replication transactions.
- The DIKTAMO recovery scheme.
- An evaluation of DIKTAMO in a simulated CXL cluster.

II. Background and Motivation

A. CXL Distributed Shared Memory

Compute Express Link (CXL) [10]–[12] is an emerging open standard high-performance interconnect. The latest versions of its *CXL.mem* protocol (CXL 3.0 and beyond) enable the compute nodes (CNs) of a cluster to directly access data in remote memory nodes (MNs) in a byte-addressable and cache-coherent manner. This unique capability enables CXL-enabled Distributed Shared Memory (CXL-DSM) systems, where coherent data sharing across nodes is supported transparently by hardware, and at cache-line granularity.

Figure 1 shows such a system with two CNs, one MN, and a CXL switch interconnecting them. In general, there are multiple MNs. The CNs have their local (i.e., private) memory hierarchies, and can access the remote memory in the MNs through the CXL switch. Physical addresses in the local memories are called *Local*, while those in the MNs are called *Remote*. On an LLC miss, routing hardware identifies if the miss is local or remote based on its target physical address. Remote misses are routed to the CXL port and access the corresponding remote directory in the MNs. Cores in a CN use write-back caching for both local and remote addresses. Data coherence is enforced with a hierarchical directory supporting a conventional cache coherence protocol like MESI: each CN has a local directory that enforces cache coherence locally; the remote directory distributed across the MNs leverages CXL to keep the remote addresses cached in the CNs coherent.

CXL is organized in three main protocol layers: a) the *Transaction Layer*, which defines the protocol transaction types and their ordering rules, b) the *Link Layer*, which ensures reliable transmission of the transaction packets, and c) the *Physical Layer*, which transmits data over the underlying interconnect.

B. CXL Reliability

In CXL-DSM clusters, faulty hardware can crash the execution of distributed shared-memory applications [33], [45], [47]. Thus, the CXL specification adds support for reliability, availability, and serviceability (RAS) [10], [17], [18]. It considers link, switch, and device (e.g., memory) failures. It detects link failures via timeouts, and retries packet transmissions to handle them. To detect data corruption, it relies on established techniques such as parity.

To contain errors, the CXL specification defines three complementary mechanisms. One is *Data Poisoning*, which allows devices to tag messages with corrupted data. A second one is *Isolation*, which allows the protocol to mark a faulty device and mask traffic from/to the device, keeping the rest of the system operational. The final one is *Viral Handling* which, when a fatal error is detected, sets a Viral_Status bit, propagating a "viral" state across all connected ports and tagging all traffic across ports. The specification [10] also introduces hardware commands to reset (re-initialize) CXL devices into a clean state after a viral handling.

This CXL RAS model has two limitations. The first one is that it does not consider processor (i.e., host) failures. In reality, handling processor failures is crucial, as CPUs are responsible for many failures in datacenters [4]. When a host fails, the dirty data it stores in its cache hierarchy is permanently lost [44].

The second limitation is that the CXL RAS model focuses only on containing errors. When a component fails, the specification tries to isolate it. There is no attempt to recover lost or corrupted data and bring the workload to a consistent state [44]. In reality, when a node participating in a shared-memory program fails and its dirty cached data is lost, continuing execution with an inconsistent state is not acceptable. The workload state must be corrected.

A possible approach to avoid data loss on a CN failure is to always use write-through writes for remote physical addresses, propagating all updates to MNs—while assuming that the MN memory is resilient [33], [45]. However, this approach can introduce substantial performance overheads, especially in systems that support the TSO memory consistency model [37], [41]. In TSO, writes from the same core to different addresses need to be fully serialized. Hence, an update from a core can only be written through to an MN after the previous update from the core has completed its round trip to an MN (i.e., request and acknowledgment).

As an example, Figure 2 considers a CXL-DSM cluster of 16 CNs of 4 cores each, and 16 MNs interconnected via a CXL switch, running several 64-threaded shared-memory parallel applications. The architecture and applications are described in Section VI. The figure shows the applications' execution times using either the MESI write-back protocol (WB) or a write through (WT) protocol. The first environment is not resilient, as dirty cached data can be lost on a CN failure. The second environment is resilient because, like in previous literature [33], [45], we assume that MN memory is fault

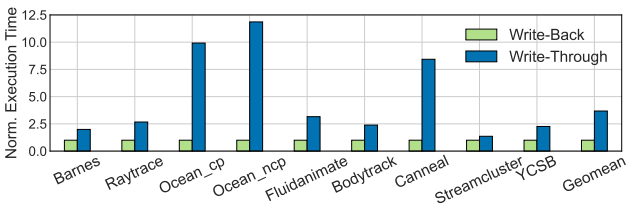


Fig. 2: Application execution time using a write-back or a write-through protocol. The bars are normalized to the former.

tolerant and that, as soon as a write updates an MN's DRAM memory, the update cannot be lost. MN memory can be made fault-tolerant through techniques like replication, persistency, or the use of more expensive technologies. How this is done is not relevant to our work.

The bars in the figure are normalized to the WB execution time. We see that WT adds substantial overhead. On average, the execution time of WT is 3.86x higher than WB's. A major reason for WT's longer execution time is that, under TSO, multiple write-through updates from a core cannot overlap with each other.

As we seek to increase the resilience of CXL, we can get inspiration from existing techniques that distributed systems use, such as replication of data records across nodes to survive node loss [1], [15], [31], [34], or data logging to recover old data versions [46], [50]. However, these techniques are typically implemented in software and operate at coarse granularity. In CXL-DSM, where data coherence is maintained at cache-line level and is enforced in hardware, we need solutions that are hardware-based, fine-grained, and low-overhead. In addition, resilient architectures that use CXL memory [20], [33] typically support transactions, which present different tradeoffs than our target general-purpose environment. Discussion on related work is presented in Section VIII.

III. Overview of DIKTAMO

In this paper, we address the two limitations of the CXL RAS model described above by extending CXL to handle the failure of a CN and to correctly recover the application state after a CN failure. We call the proposal *DIKTAMO*. DIKTAMO's extensions are mostly focused on the CXL Transaction Layer and minimally affect the CXL Link Layer.

A. Overall Operation

To accomplish its goals, DIKTAMO: 1) adds extra functionality to ordinary execution, 2) extends error detection, and 3) provides error recovery. In this section, we overview each of these three components.

1. Extensions to Ordinary Execution. DIKTAMO augments the CXL Transaction Layer for writes that target the shared CXL memory (i.e., remote writes). Any such write performed by a core in a CN is augmented with messages that propagate the update to a small set of other CNs (i.e., the *Replicas*). The replicas save the update in a local hardware *Logging Unit*. The original write cannot commit until the Logging Units in all the replicas are updated. However, crucially, multiple writes from the same core can overlap their replica update. The

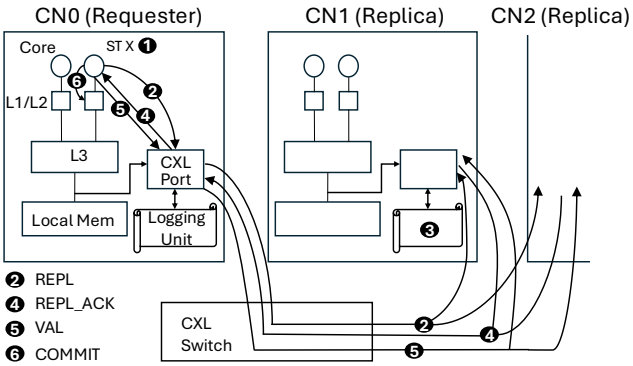


Fig. 3: DIKTAMO extensions to the transaction of a write to an address in CXL memory, assuming two replicas.

Logging Units compress the data they receive and, at regular intervals, dump it to the memory of MNs. As we describe in Section III-B, we consider that the MNs are safe from faults.

Figure 3 shows the messages added to a write ST X ① from a core in CN0 to a physical address in CXL memory when the system uses two replicas. In addition to the regular coherence protocol transaction for the write itself and before the write can commit, the protocol sends two *Replication (REPL)* messages ② with the update to the two replica nodes. The latter save the update in the log of their Logging Unit ③ and respond with a *REPL_ACK* message ④ to the originating core. The originating core waits until two events happen: it has received all the REPL_ACKs and the write coherence transaction has completed. At this point, the protocol: 1) sends two *Validation (VAL)* messages to the replicas ⑤ to notify them that replication was successful, and 2) commits the write ⑥. When a log receives the VAL, it marks the log entry as validated.

The number of replicas per update (N_r) is a design parameter. It is a small value (e.g., 1–4) that depends on the machine size, trading off fault-tolerance and overhead [13], [21], [24], [34]. To determine which CNs save the replicas for a given cache-line address, the hardware uses a hash function on the line address. Hence, all the updates to a given line address are logged in the same set of N_r nodes. It is possible that the writer of a line is in one of these N_r nodes for the line.

Loads are unaffected by DIKTAMO. The same stands for writes to physical addresses in the local memories of the CNs.

2. Error Detection. CXL's Link Layer detects errors in devices, switches, and links. DIKTAMO uses CXL's Link Layer for error detection and extends it minimally to detect CN failures. Specifically, each CXL switch includes an extra Viral_Status bit for each of the CNs that it is connected to. Additionally, a switch never responds on behalf of a failed CN, since DIKTAMO's goal is correct execution—not just isolation. Section V-A describes error detection.

3. Error Recovery. After a CN error is detected, one of the live cores is informed with a CXL MSI message [10] to act as a Configuration Manager. Such core brings the workload execution to a halt and initiates a distributed software algorithm

that disables the faulty CN and recovers the application state to the correct state immediately before the fault. Sections V-B and V-C describe the recovery.

B. Failure Model

The focus of this paper is on CN failure, leaving link, switch, and device failure handling to the support already present in the CXL specification. For CN failures, we assume the standard *fail-stop* model: a faulty CN halts and ceases participation in the protocol. This model is widely used in practical crash-fault-tolerant (CFT) systems such as RAMCloud [31], FaRM [15], and Hermes [24], as it captures the dominant failure modes in modern data centers (e.g., power loss, kernel panic, or hardware crash).

Byzantine or malicious behavior (e.g., arbitrary message fabrication or equivocation) is out of scope. This assumption is widely consistent with prior work in crash-fault-tolerant replication [15], [24], [31], where the dominant concern is reliability against hardware or software crashes rather than adversarial behavior. We also consider silent data corruption errors out of scope.

As in prior work [33], [45], we assume that MNs do not suffer faults, and that any data that is written to them is safe. MNs are bigger, more expensive servers that have stronger RAS support, possibly based on techniques such as replication, persistent storage, or the use of a more expensive technologies. How such resiliency is accomplished is not relevant to our work; we simply assume that the MN memory is safe at no additional latency cost.

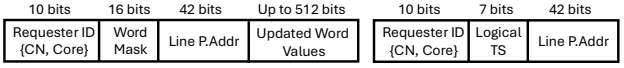
The goal of DIKTAMO is not to attain the highest cluster availability. Rather, it is to recover the program state that existed in caches, directory, and memory right before the CN failure. No committed store can be lost as long as at least one replica remains online.

IV. Augmenting Ordinary Execution

This section describes the DIKTAMO extensions added to the ordinary execution: the message types, the Logging Unit, the logical timestamps, three variations of the DIKTAMO replication protocol, and the log processing.

A. DIKTAMO Messages

DIKTAMO introduces three new protocol messages: *Replication* (`REPL`), *Replication Acknowledgment* (`REPL_ACK`), and *Validation* (`VAL`). The `REPL` message (Figure 4a) is sent by a core in a requester CN to multiple replica CNs to replicate a remote store (or a set coalesced stores to the same cache line). It carries: a) the *Requester ID*, which includes the CN and core IDs, b) a bit mask that shows which words within the line are updated by the store(s) (in this paper, we assume word granularity, but the design can be modified to use finer granularity), c) the physical address of the line, and d) the updated values of the words. The last field is of varying size, depending on the number of updates that are coalesced. The maximum size is a cache line size.



(a) REPL message (b) VAL message

Fig. 4: DIKTAMO's `REPL` and `VAL` messages.

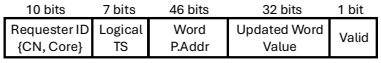


Fig. 5: DIKTAMO log entry in the SRAM Log Buffer.

The `REPL_ACK` message is sent by the *Logging Unit* of each replica CN back to the requester core. It is sent after the unit has applied the `REPL` update to its hardware log. Upon receiving all `REPL_ACK`s, the requester core knows that replication has reached all replicas.

The `VAL` message (Figure 4b) is sent by the requester core to all replica CNs after it receives all their `REPL_ACK`s. `VAL` indicates that the write is committed and so the Logging Unit should record the update as such. `VAL` carries the Requester ID and the target cache line address. It also carries a logical timestamp (TS), which is used to order the updates as they are logged in the Logging Unit (Section IV-C).

B. Logging Unit

Each CN has a hardware Logging Unit that logs the data of incoming `REPL` messages, compresses the data and then, at regular intervals, sends it to the MNs, where it is saved. The unit is linked to the CN's CXL port (Figure 3).

The Logging Unit's log is made of DRAM memory for density. To improve access speed, however, incoming `REPL` and `VAL` messages update a small SRAM memory called the *SRAM Log Buffer*. A `REPL` allocates an entry in the SRAM Log Buffer. Later, the corresponding `VAL` sets the Valid bit of the entry, and the entry becomes eligible to be moved to the DRAM log in the background. In there, it will be compressed and eventually sent to an MN. Non-VALed entries correspond to writes that have not yet committed. These entries cannot be sent to the DRAM log. Persistency of the log is not required, since DIKTAMO employs a replication factor of N_r. This allows the system to tolerate up to N_r-1 CN failures without losing state.

Figure 5 shows the layout of an entry in the SRAM Log Buffer. Each entry includes the data of a single remote store instruction. If a received `REPL` message carries multiple stores to the same line, then the Logging Unit splits the message into an equal number of log entries. Each entry has the Requester ID, a logical timestamp, the physical address and value of the updated word, and a valid bit. The latter is set only when the unit receives the `VAL` message.

C. Logical Timestamps

The DIKTAMO protocol decouples the allocation of a log entry in the SRAM Log Buffer from the saving of the entry into the DRAM log. The former is performed on reception of a `REPL`, while the latter can only happen after the reception of the `VAL`. Now, consider the case where the cores in a given

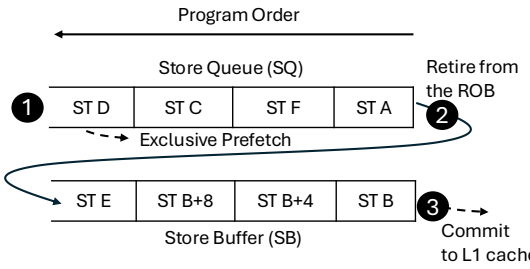


Fig. 6: Multiple stores in an out-of-order core with TSO.

CN perform two updates to the same memory location. The VALs are issued in the order of the updates—since they are issued at the point when an update commits. However, the CXL fabric could potentially reorder the two VALs and they could reach the SRAM Log Buffer out of order. To ensure that the log knows the order of the VALs, we add a 7-bit logical timestamp (TS) to the VAL messages. TS is incremented every time a VAL is sent. In the logging CN, the SRAM Log Buffer saves the VAL's TS in the log entry. However, it only pushes the entries from that word into the DRAM log in timestamp order. This ensures that the recovery will not apply the updates in the wrong order.

D. Three Variations of DIKTAMO's Protocol

In general, when a store is ready to commit, it issues a message that invalidates all other cached copies of the line and brings the line to the local cache. We call this operation the *Coherence* transaction. DIKTAMO augments the coherence transaction with a *Replication* transaction, which involves sending multiple REPLs, receiving multiple REPL_ACKs, and sending multiple VALs. Only when both the Coherence and the Replication transactions for a store complete can the store commit—i.e., merge the update into the local cache. In our design, we propose three variations of DIKTAMO that overlap the Coherence and the Replication transactions for a store differently. Before we present them, we describe the life of a store instruction.

1. The life of a Store Instruction. Figure 6 shows multiple store instructions in an out-of-order core with TSO. For ease of explanation, we use two logical structures: the store queue (SQ) and the store buffer (SB). When a store enters the reorder buffer (ROB), it also enters the SQ. When its target physical address is resolved, an exclusive prefetch is sent to memory to bring the target cache line in Exclusive state to the L1 cache of the core ①. When a store reaches the head of the ROB (and hence of the SQ), it retires into the SB ②. With the TSO memory consistency model [37], [41], stores drain from the SB in order, without overlap: a store is merged with the memory system only when it reaches the head of the SB ③. We will call this point the store commit.

A common optimization is *store coalescing*. Consecutive stores in the SB that target different words of the same cache line and are not interleaved by a store to a different line are coalesced and committed to memory in a single transaction. For instance, in Figure 6, if stores ST B, ST B+4, and ST B+8 write different addresses of the same line, they will be coalesced and merged with memory together. In such cases, DIKTAMO sends a single REPL for all the coalesced stores. In the REPL (Figure 4a), the Word Mask sets the corresponding bits and the Values field includes all the updated word values. This reduces the network traffic of replication. All three DIKTAMO designs support coalescing.

If the exclusive prefetch for a store successfully brings the target line into the L1 cache and the line has not been invalidated, downgraded or evicted by the time the store reaches the head of the SB, then what we call the Coherence transaction takes no time. However, for illustration purposes, in the following discussion, we will assume that the line has been invalidated, downgraded, or evicted and, when the store reaches the head of the SB, the Coherence transaction experiencies the full latency. In cases where the exclusive prefetch has been issued but not completed by the time the store reaches the head of the SB, the Coherence transaction will only observe partial latency.

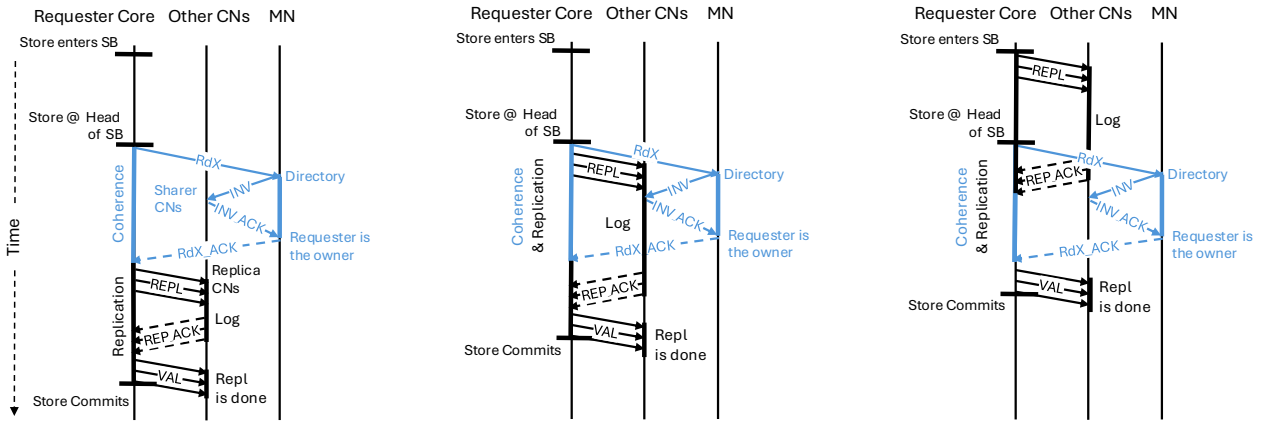
2. First Design: DIKTAMO-baseline. In this design, the Replication transaction of a store starts only after the Coherence transaction has completed. When the Replication transaction completes, the store can commit.

Figure 7a shows the timeline of a store from the time it enters the SB in the requester core until it commits. It includes the operations in the requester core, in other CNs, and in the MN that contains the directory. It shows the Coherence transaction in blue and the Replication one in black. We see that, when the store reaches the head of the SB, the Coherence transaction starts. We show the general case where the line is currently cached in Shared state by other nodes. A request for ownership (RdX) is sent, which reaches the directory in an MN. The latter sends invalidations to all sharer CNs, collects their ACK responses, marks the requester as the owner, and responds with the cache line (RdX_ACK).

The Replication transaction starts now. The CN with the requester core sends REPL messages to replica CNs (three in the example). Each replica CN logs the update and replies with a REPL_ACK to the requester. When the requester receives all the REPL_ACKs, it sends the VALs to all the replicas and commits the store—i.e., it drains the store from the SB and updates the cache line in L1.

3. Second Design: DIKTAMO-parallel. In this design, the Replication transaction of a store starts at the same time as the Coherence transaction when the store reaches the head of the SB. Both transactions overlap in time. Figure 7b shows the timeline of a store. Note that DIKTAMO-parallel can only send the VALs (and immediately commit the store) when two conditions are met: 1) the originating core has received all the REPL_ACKs and 2) the Coherence transaction has completed. The advantage of this design is that, by overlapping the two transactions, the store commits sooner.

4. Third Design: DIKTAMO-proactive. In both DIKTAMO-baseline and DIKTAMO-parallel, replication is initiated no earlier than when a store reaches the head of the SB. As a



(a) DIKTAMO-baseline (b) DIKTAMO-parallel (c) DIKTAMO-proactive

Fig. 7: Timeline of the Coherence and Replication transactions of a store in the three variations of the DIKTAMO protocol. This timeline assumes that an early exclusive prefetch of the line targeted by the store has been ineffective because the line has since been invalidated, downgraded, or evicted from L1. In cases where the exclusive prefetch has been issued but not completed by the time the store reaches the head of the SB, the Coherence transaction will only observe partial latency. If the exclusive prefetch has completed and the data is in the L1, the Coherence transaction will take no time.

result, replication is *serialized* across stores. However, all the necessary information for the replication is known at least as soon as the store retires and enters the SB (② in Figure 6). To send the REPLs, we do not need to wait for the many cycles that a store may wait in the SB until it reaches the SB head [26].

To avoid waiting in the SB, we introduce DIKTAMO-proactive, where the REPLs for a store are sent out as soon as the store retires and enters the SB. Figure 7c shows the store timeline. As before, DIKTAMO-proactive can only send the VALs (and immediately commit) when both the REPL_ACKs are received and the Coherence transaction has completed.

Sending the REPLs earlier (e.g., when the exclusive prefetch is issued) is undesirable because the store is speculative. As a result, the store could be squashed due to a misspeculation, which would lead to two issues: (a) unnecessary network traffic, and (b) nonVAL-ed entries in logging units that would require a dedicated mechanism to remove. Unlike an exclusive prefetch, a squashed Replication transaction does not have a useful side effect.

DIKTAMO-proactive has two performance advantages. First, it hides the latency of replication behind the existing SB queuing delays. *Second and crucially, it overlaps the REPL–REPL_ACK cycles of multiple retired stores waiting in the SB, delivering faster execution without violating TSO*. To see how, Figure 8 shows the case of stores ST A, ST B, and ST C waiting in the SB. The leftmost chart shows the timeline for DIKTAMO-parallel. Each store waits until it reaches the head of the SB and then performs the Replication transaction in sequence. In contrast, the rightmost chart shows the timeline for DIKTAMO-proactive. It issues REPLs for consecutive stores in order, but allows their REPL–REPL_ACK cycles to fully overlap. DIKTAMO-proactive can support as many outstanding replication requests as the size of the SB,

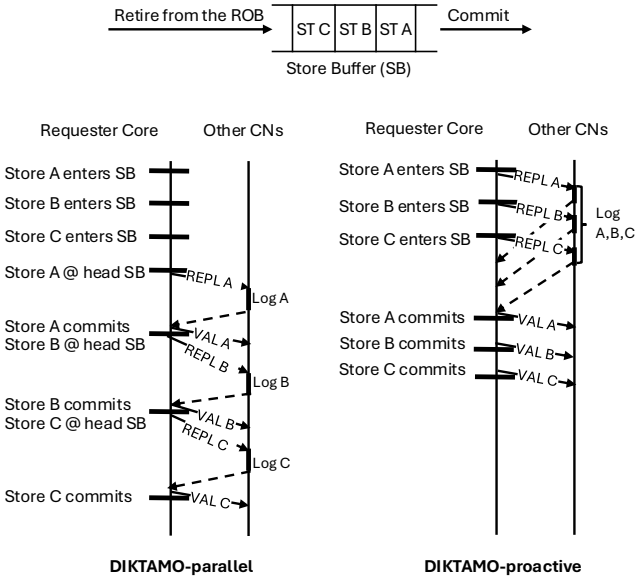


Fig. 8: DIKTAMO-proactive overlaps the REPL–REPL_ACK cycle for multiple retired stores waiting in the SB.

without violating TSO.

5. DIKTAMO-proactive and Store Coalescing. DIKTAMO-baseline and DIKTAMO-parallel trivially support store coalescing: as the two (or more) oldest stores in the SB are coalesced, only a single REPL message set is sent for all of them. However, DIKTAMO-proactive requires some subtle redesign to support coalescing. This is because individual stores send their own REPL sets as soon as they retire and are placed in the SB, while, later on, coalescing may potentially merge multiple of them when they are draining from the SB. The result will be a single multi-store commit operation and, hence, a single VAL. This would be confusing and require

supporting special cases.

Consequently, to support coalescing, we slightly modify DIKTAMO-proactive as follows. When a store B is deposited into the SB, it never issues its `REPL` set. If it finds that it is immediately preceded by a store A to another word of the same line, it is coalesced with A; no other action is taken. Later, when the next store C is deposited into the SB, if C cannot be coalesced with B, B issues its `REPL` set (which carries B and all the earlier stores coalesced with B). Finally, if a store B reaches the head of the SB without having sent its `REPL` set, it sends it at this point (carrying B and all the stores coalesced with B). This design supports coalescing, while initiating only a single `REPL`s-`REPL_ACK`s-`VAL`s transaction for each commit.

E. DIKTAMO's Log Processing

The Logging Units of all the CNs continuously compress the logged data and, at periodic intervals, they all save the compressed data to the MNs, where the data is safe from faults. Given that there are N_r replicas, groups of N_r nodes have the same data in their logs. We call these groups of CNs *Replica Groups*. To reduce overhead, the Logging Units in a Replica Group divide their work. Specifically, in a Replica Group, each of the Logging Units is in charge of saving only log entries within a range of physical addresses.

The process followed by each Logging Unit is as follows. The hardware reads the DRAM log in sequence, from older to newer entries, extracting the log entries that this particular Logging Unit is in charge of saving (based on the physical address). As these log entries are being extracted, the hardware compresses them to save memory space. For compression, we use gzip [14] with compression level 9, which attains an average compression factor of 5.8x in our experiments. At regular intervals, the Logging Units synchronize through the controllers in the MNs, save their compressed logs in 64-byte messages to the MNs, synchronize again, and clear the logs.

The size of the log and the interval between flushes are set so that logs typically do not overflow. In rare cases, it is possible that a log in a node reaches a high watermark prior to reaching the time to flush the log. In this case, the Logging Unit in the node triggers an Early Flush interrupt to the Logging Units of all the other nodes. Then, all the Logging Units synchronize and perform the flushing operation.

V. DIKTAMO and Failures

This section describes how DIKTAMO detects a CN failure and outlines the DIKTAMO recovery.

A. Failure Detection Mechanism

The Link Layer in CXL detects both transient and permanent errors in devices, switches, and links [10]. DIKTAMO reuses this support for all the messages it adds to the protocol. However, since the Link Layer does not support CN failures (Section II-B), DIKTAMO extends it as follows. First, the CXL specification indicates that CXL switches contain Viral_Status bits for various ports to indicate failure conditions in those ports [10]. DIKTAMO extends the switch design to have one Viral_Status bit for each CN that is connected to the switch. When the switch detects that a given CN is unresponsive, it sets the corresponding Viral_Status bit.

Additionally, the CXL specification indicates that, when a message is sent to a failed device, the device or nearest switch responds with a message with *poisoned* or *viral* data [10]. This is done so that the failed device is isolated. However, in an environment where multiple CNs share data, one cannot afford responding with incorrect data, as it could pollute the state of the running application. Consequently, in DIKTAMO, the switch connected to a failed CN will not respond at all to requests to the failed node (and the latter will not respond either because of the failure model we assume).

Instead, as soon as a switch detects a failed CN, it sends a Message Signaled Interrupt (MSI) to a core in one of the live CNs to trigger a recovery. An MSI is the standard mechanism used by CXL devices to signal interrupts to a host CPU [10]. They function identically to how they are used in PCI Express (PCIe). The receiver core, which we call the *Configuration Manager* (CM) for this fault, will trigger a coordinated pause of all the CNs, and then initate a software-driven recovery.

B. Recovery Protocol Overview

The objective of the software-driven coordinated recovery is to reconstruct the consistent state that the application had immediately before the fault, and restart the application on the remaining functioning CNs. The process involves accessing the directory state in the MNs and identifying the memory lines that the directory records as being present in the caches of the failed CN, either in state `Shared` or `Modified`. For the former, the directory entry must be updated by removing the failed CN as a sharer. For the latter, the logs in the MNs and in the Logging Units must be searched to find the latest update before the crash. Then, such update must be applied to memory, and the directory entry must be changed to indicate that there are no sharers. These operations are performed by recovery software. Our main concern is to guarantee correctness and minimize complexity.

Figure 9 shows the messages exchanged by the nodes during recovery. Table I describes the messages. These messages can leverage the mailbox interfaces in the CXL RAS API (v0.8) [18], which enable host–device communication for reliability and error-handling events. The recovery process begins with the CM broadcasting an `Interrupt` to all live CNs in the cluster. These messages are directed to all the CPU cores in the CNs and to the per-CN Logging Units. Upon receiving the interrupt, cores and Logging Units complete all outstanding requests/operations, and pause. Then, they respond with an `InterruptResp`. Once all `InterruptResp`s have been received, the CM issues an `InitRecov` to all MNs, prompting software handlers in the directory controllers of the MNs to initiate recovery. The actions followed by these handlers set the directories and memories to a consistent state. They are described in Section V-C.

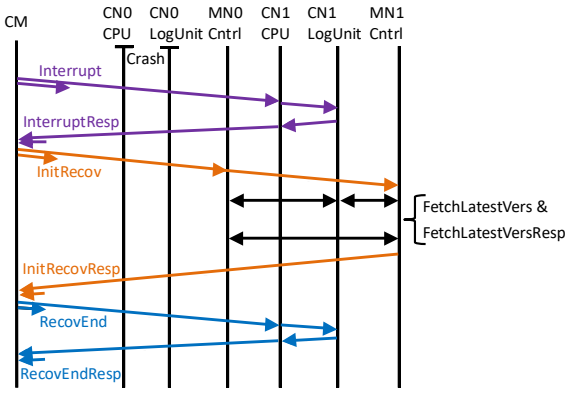


Fig. 9: Message exchange during recovery in DIKTAMO.

TABLE I: Recovery protocol messages.

Message	Definition / Usage
`Interrupt`	Sent by the CM to all live CNs and their Logging Units to complete outstanding requests and other operations in the cores and in the Logging Units.
`InterruptResp`	Sent by CNs to confirm completion of outstanding requests and operations.
`InitRecov`	Issued by the CM to all MNs to initiate directory-level recovery actions by software handlers.
`FetchLatestVers`	Request from the directory controllers to the Logging Units and MNs for the latest logged updates to the lines that were owned by the failed CN's caches.
`FetchLatestVersResp`	Response with the latest logged updates to the lines that were owned by the failed CN's caches.
`InitRecovResp`	Issued by handlers in directory controllers to acknowledge completion of directory-level recovery actions.
`RecovEnd`	Broadcast by the CM to all live CNs and their Logging Units to signal that recovery is complete and they can resume execution.
`RecovEndResp`	Sent by CNs to acknowledge completion of recovery and execution resumption.

Once the directory and memory in an MN have been brought to a consistent state, the handler in the directory controller sends an *InitRecovResp* to the CM. Upon the reception of *InitRecovResp* from all live directory controllers, the CM broadcasts a `RecovEnd` to all CNs and Logging Units, signaling the completion of the recovery protocol. Each node responds with a `RecovEndResp` and resumes program execution.

C. Recovery at the Directory Controller

Each directory controller runs a software handler that repairs the directory and memory state in the local MN. The high-level algorithm is shown in Algorithm 1. The handler first traverses the local directory and performs two operations. For the directory entries where the failed CN was a sharer, it removes the failed CN as a sharer. For the directory entries where the failed CN was the owner, it collects the addresses of all the words in the corresponding lines in a list called `Owned`.

Algorithm 1: Recovery of the Directory and Memory State by a Software Handler at the Directory Controller

Input: Set of dir entries marked as owned or shared by failed CN
Output: Corrected directory and memory state
For all dir entries where the failed CN was a sharer:
  Remove the failed CN as sharer;
For all dir entries where the failed CN was the owner:
  Collect word addresses of corresponding lines in `Owned` list;
Send `FetchLatestVers` with `Owned` list to relevant LUs;
Collect all `FetchLatestVersResp`;
Identify LatestVersions in all `FetchLatestVersResp`;
Remove addresses of identified LatestVersions from `Owned` list;
If `Owned` list not empty:
  Send `FetchLatestVers` with `Owned` list to relevant MNs;
  Collect all `FetchLatestVersResp`;
  Identify LatestVersions in all `FetchLatestVersResp`;
Write identified LatestVersions from logs in MNs and LUs to mem;
Mark their directory entries as not owned;

The handler then proceeds to collect the latest versions of all the words in `Owned`. This is done in two steps. First, the handler sends a `FetchLatestVers` message with the `Owned` list to all the Logging Units that could have updates to such words. In each Logging Unit, a handler searches for them and returns a `FetchLatestVersResp` message. The message contains, for each requested word address, the latest logged update to it. When the directory receives the `FetchLatestVersResp` from all replicas, it compares them. For each address in the `Owned` list, two cases are possible. One case is when all the replica logs have an update (which is the same in all replicas). If so, this is the latest version. A second case is when none of the replica logs have an update. In this case, the version is in the MN logs.

In the second step, the directory controller removes from the `Owned` list all the addresses for which it has collected the latest versions. It then sends a `FetchLatestVers` message with the `Owned` list to all the MNs whose log could have the missing versions. In each MN, a software handler searches the log and returns the latest version of the addresses in a `FetchLatestVersResp`. When the directory controller handler receives the `FetchLatestVersResp` from all replicas, it finds the latest versions of the remaining words. Then, the handler collects all the latest versions found in the two steps, writes them to memory in the local MN, and updates the corresponding local directory entries so the lines are not owned by any CN.

VI. Methodology

Architecture and Simulation Infrastructure. We simulate a distributed system with 16 compute nodes (CNs) of four cores each, and 16 memory nodes (MNs), interconnected via high-bandwidth CXL links. The architecture parameters are shown in Table II. There is a single CXL switch connecting all CNs to all MNs. Cores are out-of-order with private L1/L2 caches. All cores in a CN share an inclusive L3 cache. Logging units operate at 500 MHz. Each Logging Unit has a 4KB SRAM Log Buffer and a 18 MB DRAM log. We model CXL using parameters from the literature [25], [29], [30]. We assume

TABLE II: Architecture parameters. RT means round trip.

System Configuration	
Compute Nodes (CN); Memory Nodes (MN)	16 CNs; 16 MNs
CPU Frequency; Logging Unit Frequency	2.4 GHz; 500 MHz
Cores per CN	4 Out-of-Order cores
Load Queue Size; Store Queue Size	128; 72
Cache Hierarchy & Local Memory (per CN)	
L1 Cache (Size; Associativity; RT Latency)	48 KiB; 12-way; 5 cycles
L2 Cache (Size; Associativity; RT Latency)	512 KiB; 8-way; 13 cycles
L3 Cache (Total Size; Associativity; RT Latency)	8 MiB; 16-way; 36 cycles
Cache Line Size	64 bytes
CXL Memory and Interconnect	
DRAM Access Latency	45 ns
CXL Switch Bandwidth (Aggregate)	4 TB/s [29]
CXL Network RT Latency (includes Dir Lookup)	200 ns [25]
Logging Unit	
Log Buffer Size	SRAM: 4KB (4ns access); DRAM: 18MB
Period between Log Dumpings	2.5 ms
Replication Factor	N_r=3 copies

that the MNs have features that make them fault tolerant. We follow prior work [33], [45] in this assumption.

We use the Structural Simulation Toolkit (SST) [36] to model the architecture. Execution traces of the program execution are collected per core using Pin [22], and fed into the simulator. The traces include all data accesses, including synchronizations (lock acquire request, lock release, barrier arrive, and barrier leave). In between every two consecutive entries, the trace also includes K_i, the number of other types of instructions executed between the two accesses. In the simulations, each core advances time by adding K_i cycles in between memory operations (which assumes that non-memory instructions have a CPI equal to 1). We ensure that the synchronizations are correctly modeled. Specifically, only one thread can be inside a given critical section at a time (the others can be spinning), and threads spin on a barrier until all other threads have arrived. We compile the applications with -O3 to eliminate unnecessary local accesses.

Applications. We use a mix of applications from PARSEC [6] and SPLASH-2 [5], [42], and a custom key-value workload using Yahoo! Cloud Serving Benchmark (YCSB) [9], [35]. From PARSEC, we use *bodytrack*, *fluidanimate*, *streamcluster*, and *canneal*, which are a range of animation, computer vision, engineering, and data mining workloads. From SPLASH-2, we use *raytrace*, *barnes*, *ocean_ncp*, and *ocean_cp*, which are workloads that stress memory-level parallelism. For YCSB, we use a distributed key-value store backed by a hashtable, inspired by Bigtable [7].

All applications are multi-threaded and are executed with 64 threads running on all cores and fully utilizing the cluster. For each application, we execute 6.4B instructions or more. For the PARSEC and SPLASH-2 applications, we skip the initialization and start the evaluation when the application reaches the parallel section or the Region of Interest. We use the Medium size of the applications or larger. We place the shared data on the CXL memory, and the private data of the threads on the local memory of the CNs

For YCSB, the key value store has 1.6M records of size 1KB. The application utilizes a database organized in an array-like format. It performs 6.4 million put or get record operations to the CXL memory. It issues 80% reads and 20% writes with a uniform record access distribution. We place all the key value store data on the CXL memory.

Configurations. We evaluate five architecture configurations that handle the accesses to the data allocated in CXL memory (i.e., remote data) differently.

- **Write-Back (WB)**: Conventional, non-resilient system where stores to remote data (i.e., remote stores) are committed to local caches without any replication and without simultaneously saving the update in any MN. This configuration is not resilient: the failure of a CN results in data loss. It provides a lower-bound execution time.
- **Write-Through (WT)**: Resilient system where remote stores are write through, updating the memory in a MN. As soon as a write updates a MN's DRAM memory, we assume that the update cannot be lost—since the MN is fault tolerant. Under the TSO memory model [37], [41], multiple write-through updates from a core cannot overlap with each other.
- **DIKTAMO-baseline**: Baseline version of our resilient write-back DIKTAMO protocol, where remote stores are replicated to N_r CNs after their coherence transactions complete.
- **DIKTAMO-parallel**: Variant that optimizes DIKTAMO-baseline by overlapping the issue of the replication and coherence transactions of a store.
- **DIKTAMO-proactive**: Proposed DIKTAMO variant, which improves on DIKTAMO-parallel by allowing the initiation of the replication transactions for remote stores as they enter the SB.

All DIKTAMO configurations use the same scheme for selecting replica CNs: a hash function maps the line address to a Replica Group. N_r is set to 3, so that every remote store is replicated in three distinct Logging Units across the cluster. This number is very conservative (i.e., large) for our small 16-CN machine, given that it is similar to the number used in prior works evaluating datacenter-scale distributed systems [13], [38]. We use this number so we can model the additional traffic that DIKTAMO would add in a large machine.

VII. Evaluation

In this section, we first evaluate the performance of DIKTAMO, then characterize DIKTAMO, evaluate recovery, and finally perform sensitivity analyses.

A. Performance

Figure 10 shows the execution time of our workloads when we run them with the five different protocols. We normalize execution time to the *WB* configuration, which has high performance but is intolerant to faults.

We observe that WT is prohibitively expensive, especially for write-intensive workloads such as Ocean-cp and Ocean-ncp. Writing through remote writes can be very slow in TSO, as writes from the same core cannot overlap. As a result, the SB can fill-up, which results in the stall of the core execution. The average execution time of WT over WB is 3.68x.

DIKTAMO-baseline serializes the Coherence and Replication transactions for a remote store. We see that it is

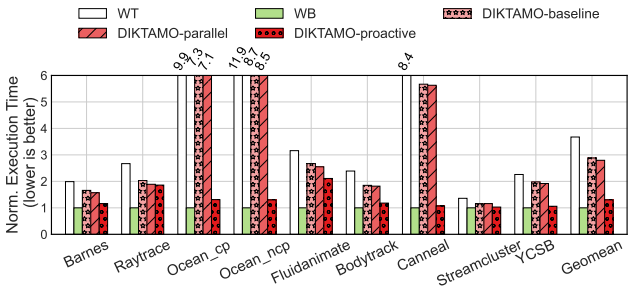


Fig. 10: Execution time with different protocols.

still significantly slower than WB. On average, its execution time is 2.88x higher than WB. DIKTAMO-baseline penalizes write-intensive applications by slowing down SB draining. However, because of its write-back approach, it is a faster and resilient alternative to WT.

DIKTAMO-parallel overlaps the Coherence and the Replication transactions for a remote store. However, we observe that the benefits over DIKTAMO-baseline are minor (only 3% on average). The reason is that the architecture we model is equipped with exclusive prefetching. As soon as a store's target address is known, a request for ownership is sent to the memory system. Thus, when the store reaches the head of the SB, the Coherence transaction has often completed, exposing the Replication transaction like in DIKTAMO-baseline.

DIKTAMO-proactive proactively initiates the Replication transaction for a remote store as soon as the store enters the SB. The advantage of this technique is twofold. First, it overlaps the Replication transaction with the store's waiting time in the SB. Second, it *enables the overlapping of the Replication transactions for the stores in the SB*. As a result, this design substantially boosts performance. The execution time with DIKTAMO-proactive is on average only 30% higher than WB, while making the protocol resilient to failures.

Figure 10 shows that raytrace and fluidanimate do not benefit much from DIKTAMO-proactive. To understand why, Figure 11 shows the fraction of DIKTAMO-proactive REPLs that are sent when the store is at the head of the SB. This is the latest time that DIKTAMO-proactive can send the REPLs. If it sends them at this time, there is no difference between DIKTAMO-parallel and DIKTAMO-proactive. The higher this fraction is, the lower the difference between DIKTAMO-proactive and DIKTAMO-parallel is. We see that raytrace and fluidanimate have a high fraction, which explains their behavior in Figure 10. In these applications, a store often remains in the SB without sending REPLs, with the hope of coalescing with a future store that never appears.

Streamcluster also has a high fraction in Figure 11. However, because the fraction of requests that are remote writes is relatively small, all protocols behave similarly (Figure 10).

B. Characterization Analysis

We now analyze several aspects of DIKTAMO-proactive.

Impact of Store Coalescing on DIKTAMO-proactive. As discussed in Section IV-D.5, supporting store coalescing in DIKTAMO-proactive required a design that sometimes delays

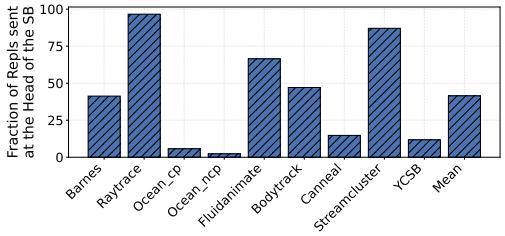


Fig. 11: Fraction of REPLs sent by DIKTAMO-proactive when the store is at the head of the SB.

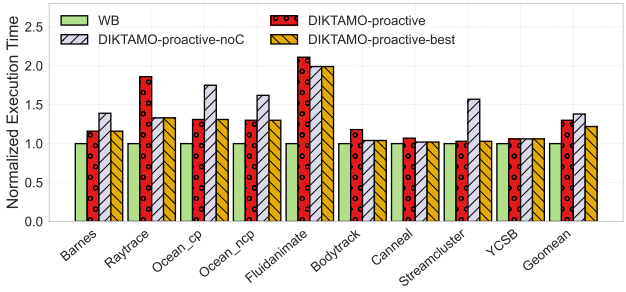


Fig. 12: Evaluating the impact of store coalescing on DIKTAMO-proactive.

sending REPLs for stores in the SB. Hence, supporting coalescing affects performance in two ways. On the one hand, it can improve performance by combining multiple stores and reducing the number of write transactions. On the other hand, it may delay sending REPLs for some writes, reducing the ability to hide the Replication transaction.

Figure 12 examines whether the positive or the negative effects of supporting coalescing dominates. It repeats Figure 10 for WB and DIKTAMO-proactive (which supports coalescing), and adds two more bars: DIKTAMO-proactive-noC (which does not support coalescing) and DIKTAMO-proactive-best, which is simply the best of DIKTAMO-proactive and DIKTAMO-proactive-noC.

Comparing DIKTAMO-proactive with DIKTAMO-proactive-noC shows that DIKTAMO-proactive is only slightly better, and that applications vary widely. Some applications like ocean-cp benefit from supporting coalescing, while others like raytrace lose performance. In general, it is hard to correlate Figure 12 with Figure 11: Figure 11 simply shows whether the REPLs were delayed to their latest times; it does not show when the REPLs were sent or whether fruitful coalescing occurred. Still, we see that raytrace, which delayed many REPLs in Figure 11, ends up being hurt by supporting coalescing in Figure 12. Finally, the lower execution time of DIKTAMO-proactive-best suggests that selectively enabling coalescing on and off may be interesting. DIKTAMO-proactive-best's average overhead over WB is only 22%, while DIKTAMO-proactive's is 30%.

Log Size. Each Logging Unit has a 4KB SRAM Log Buffer for pending transactions and an 18MB DRAM Log Buffer to log updates. Figure 13 shows the maximum uncompressed log size generated in any CN in DIKTAMO-proactive between

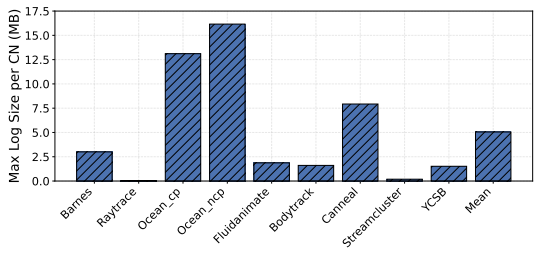


Fig. 13: Maximum uncompressed log size generated in any CN in DIKTAMO-proactive between consecutive log dumps.

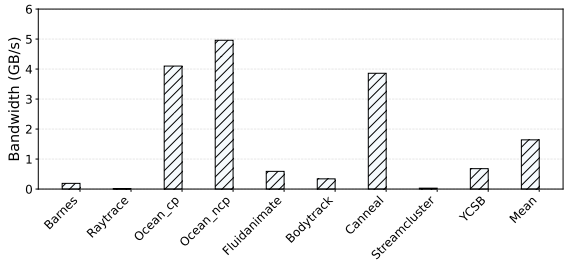


Fig. 14: Average CXL network bandwidth consumed by the 16 CNs periodically saving the compressed logs in DIKTAMO-proactive.

consecutive log dumps. This is the biggest possible DRAM buffer size needed. In practice, a much smaller space is needed, since the data is being compressed as it is being deposited into the DRAM buffer, before being flushed to MNs at the end of the period between log dumps. However, to be conservative, we size the DRAM buffer in each Logging Unit to be 18MB.

CXL Network Bandwidth Consumed by Saving the Logs. Figure 14 shows the average CXL network bandwidth consumed by the 16 CNs periodically saving the compressed logs to the MNs in DIKTAMO-proactive. We see that the bandwidth consumption is minimal and not higher than 5 GB/s across all the workloads. Hence, the execution time of workloads is largely unaffected by the log saving operations.

Lines Owned by a Crashed Node. As part of the recovery, a handler traverses the directory and finds the addresses of all the lines that were Shared or Owned by the caches in the crashed CN. Owned lines can be Dirty or Exclusive in caches, but the directory does not distinguish between them. For the Owned lines, the handler then traverses the logs to recover their value before the crash.

In this experiment, we trigger the crash of CN 0 at 12.5ms of execution in the applications under DIKTAMO-proactive, and count the number of Owned lines. The directory gives us a number, and then we check the caches of the crashed CN to count those that are indeed Dirty. We label the difference between the two counts as Exclusive, although some of them may have been evicted silently from the caches. Figure 15 shows the result for the applications. For reference, the maximum number of different lines in the caches of a CN is 163K. The figure shows that, on average, less than 29K lines are Owned, although, in YCSB, such number reaches 119K.

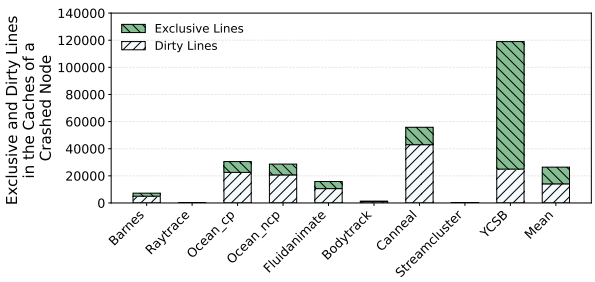


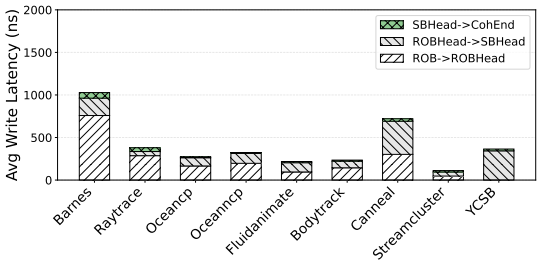
Fig. 15: Number of Exclusive and Dirty lines in the caches of a crashed CN in DIKTAMO-proactive.

Remote Write Latency Breakdown. Figure 16 shows the average latency of a write to remote data broken down into its components: *ROB→ROBHead* is the time from when the store is deposited into the ROB until it retires; *ROBHead→SBHead* is the time from store retirement until the store reaches the head of the SB. *SBHead→CohEnd* is the additional time it takes for the coherence transaction to complete, bringing the line in state Exclusive to L1; and *Replication Time* is any additional time beyond CohEnd that the write needs to remain in the SB waiting for the Replication transaction to complete. After *SBHead→CohEnd* and *Replication Time*, the write commits. Results are shown for WB (a), *DIKTAMO-proactive* (b), and WT (c).

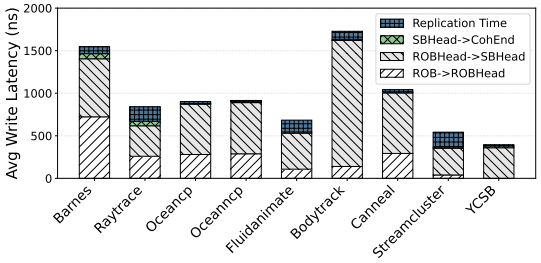
We see that, in WB (Figure 16a), *ROB→ROBHead* and *ROBHead→SBHead* dominate. *SBHead→CohEnd* is minimized by exclusive prefetching and *Replication Time* does not exist. Moving to *DIKTAMO-proactive* (Figure 16b), *ROB→ROBHead* remains mostly the same, while *ROBHead→SBHead* increases in all cases and becomes dominant. The average write spends more time in the SB because of the increased number of operations performed in a write transaction. In addition, *Replication Time* appears in several applications. The applications that have the highest increase in average remote write latency are bodytrack and streamcluster.

Correlating these write latency increases with the increases in execution time in Figure 10 is not easy, since not all the remote write latency is visible in the execution's critical path, and there are many factors that affect execution time. What is clear is the correlation between the *Replication Time* in Figure 16b and the fraction of `REPL`s sent at the SB head shown in Figure 11. Applications with a high fraction of `REPL`s sent at the SB head in Figure 11 (raytrace, streamcluster and, to a lesser extent, fluidanimate, barnes, and bodytrack) have a high *Replication Time* in Figure 16b.

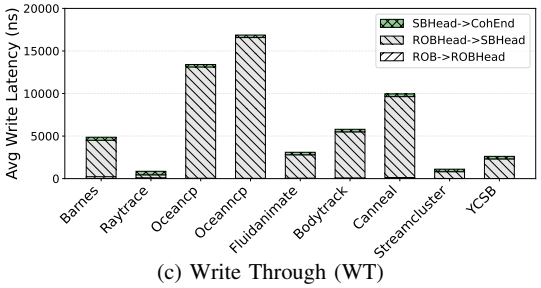
Figure 16c shows the data for WT. Note that the Y axis in Figure 16c spans a range $10\times$ larger than the other subplots. Unlike *DIKTAMO-proactive*, WT has no replication component. On the other hand, *ROBHead→SBHead* entirely dominates the latency. This is because, under TSO, WT stores often wait substantially for older stores to complete their round trip to the MN before draining from the SB. Across applications, we see a pattern like in Figure 13, which shows that applications with heavier remote-store activity also tend to require larger DIKTAMO logs. Overall, the figure shows



(a) Write Back (WB)



(b) DIKTAMO-proactive



(c) Write Through (WT)

Fig. 16: Breakdown of the average latency of a write to remote data for three different protocols. Note that the Y axis of the WT subplot (c) spans a range 10× larger than the other subplots.

why WT is costly: it turns remote-write intensity into long SB waiting time, whereas *DIKTAMO-proactive* keeps WB caching and overlaps replication with existing SB delays.

C. Recovery Latency

This section evaluates the recovery latency of DIKTAMO. Based on Section V, we break down the operations during recovery into the seven steps in Table III. `RecBegin` and `RecEnd` capture the synchronization overhead required to first start and finally end the recovery. `CM↔MN` is the exchange of recovery-control messages between the Configuration Manager (CM) and the MN directory controllers. `TravDir` is the traversal of the directory state to identify cache lines affected by the failed CN. `MN↔LU` is the exchange of information between MN directory controllers and Logging Units (LUs). `TravLU` is the search for candidate replicas of the affected cache lines in the LUs. Finally, `MNMerge` is identifying the correct version of each lost cache line (either from those received from LUs or from searching the logs already in MNs) and saving them to memory. The recovery is the same for all the DIKTAMO flavors. In this experiment, we crash one node after 12.5ms of execution and recover the system.

TABLE III: Steps of the recovery in DIKTAMO.

Step	Description
`RecBegin`	Upon detection of the failure, the CM interrupts the surviving CNs, and waits until they quiesce, complete pending transactions, and acknowledge.
`CM↔MN`	The CM sends recovery-start messages to the MN directory controllers. Once the MN directory controllers complete their recovery, they send acknowledgments to the CM.
`TravDir`	Each MN directory controller scans its directory to identify entries for cache lines shared or owned by the failed CN. For each shared line, remove the CN from the sharer list. For each owned line, save the address of the line in a "failed-CN-owned lines" list.
`MN↔LU`	The MN directory controllers send the "failed-CN-owned lines" list to the Logging Units. Later, the Logging Units return the latest versions of the lines in the "failed-CN-owned lines" list (if they had them).
`TravLU`	Each Logging Unit scans its log for validated entries matching the lines in the "failed-CN-owned lines" list and compiles a list with the most recent version of each line that it can provide.
`MNMerge`	Each MN directory controller identifies the correct versions of the lost cache lines (either from those received from LUs or from searching the logs already in MNs) and saves them to memory. The directory entries are updated accordingly.
`RecEnd`	The CM broadcasts recovery completion to the surviving CNs and waits for acknowledgments.

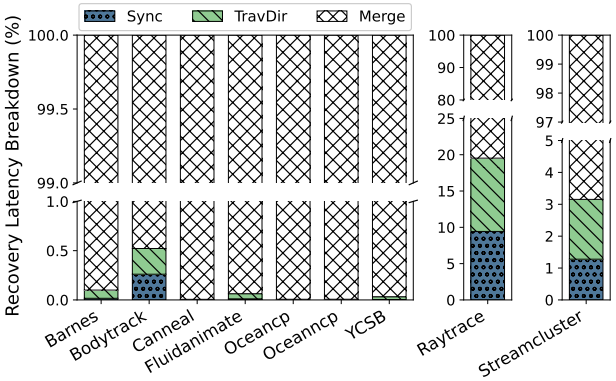


Fig. 17: Normalized latency of the recovery in DIKTAMO broken down into three main components.

Figure 17 shows the normalized latency of the steps of the recovery. We show three separate plots because of the different behavior of the applications. In each plot, the Y-axis is broken into two pieces to ensure all categories are visible. We are interested in comparing three components. The first one is the time for the CNs, MNs, and LUs to synchronize and communicate. We call it `Sync` and includes `RecBegin`, `CM↔MN`, `MN↔LU`, and `RecEnd`. The second component is the traversal of the directories. This is the `TravDir` time. The third component is finding the lost cache line versions by searching the logs in the Logging Units or in the MNs, and merging them to memory. We call it `Merge` and includes `TravLU` and `MNMerge`. From the figure, we see that, in most cases, the very large fraction of the time is spent in `Merge`—finding the lost versions and merging them to memory. This is costly because it involves substantial searching. In comparison, traversing the directory (`TravDir`) and communicating/synchronizing (`Sync`) take much less time. `TravDir` is sped-up by sequential prefetching.

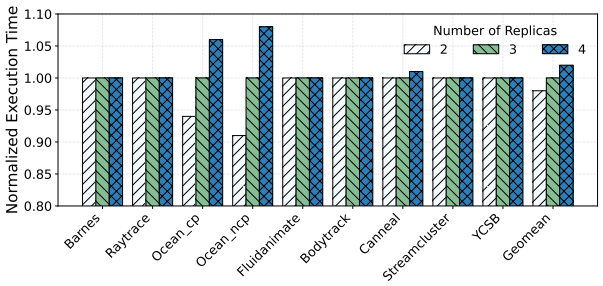


Fig. 18: Execution time with DIKTAMO-proactive varying the replication factor N_r. All bars are normalized to N_r=3.

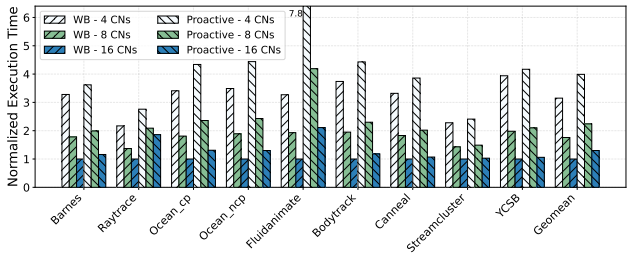


Fig. 19: Execution time of DIKTAMO-proactive and *WB* for different numbers of CNs. Bars are normalized to WB with 16 CNs.

D. Sensitivity Analyses

In this section, we perform two sensitivity analyses.

Replication Factor. Figure 18 shows the execution time of the applications with DIKTAMO-proactive for replication factors N_r equal to 2, 3, and 4. All bars are normalized to the default N_r=3. On average, the execution time increases with N_r. A system with N_r=4 takes 2% longer to execute than one with N_r=3, while one with N_r=2 is 2% faster than it. Remote write–intensive applications like ocean-cp and ocean-ncp are significantly affected. From this data, it can be shown that DIKTAMO-proactive with N_r=2 takes 27% longer than WB to execute our workloads.

Scalability with the Number of CNs. Past work suggests that CXL systems with 16 CNs are a good design point [25], [27], and some CXL switches support 32 ports [43] (which is ideal for our 16 CN and 16 MN system). Figure 19 shows the execution time of DIKTAMO-proactive and *WB* as we change the number of CNs from 4 to 16. The bars are normalized to WB for 16 CNs. The figure shows that, across all applications, DIKTAMO-proactive has good scalability. On average, going from 4 to 16 CNs, reduces the execution time by 3.15x and 3.07x for *WB* and DIKTAMO-proactive, respectively.

VIII. Related Work

Related uses of CXL. Several systems have leveraged CXL for memory expansion and pooling. CXL-ANNS [23] uses CXL memory to scale approximate nearest neighbor workloads. Memstrata [51] introduces a memory allocator in the hypervisor and the first hardware-managed memory tiering for CXL. Pond [25] addresses memory stranding issues by enabling flexible pooling.

With CXL 3.0, memory sharing has enabled new distributed system abstractions. HydraRPC [27] implements low-latency RPC over shared CXL memory. CXLfork [2] enables fast process forking over CXL shared memory. TrEnv [19] explores the use of shared memory over CXL to accelerate serverless functions.

Tigon [20] and CTXNL [40] are transactional systems over CXL. They both use software cache coherence for the majority of the data in the program and CXL-supported hardware cache coherence for metadata. Both works provide little or no discussion on recovery. Tigon uses conventional database transaction logging and recovers by concurrently restoring the log. CTXNL says it "relies on the transaction processing frameworks to ensure data consistency". Hence, recovery in these systems is software-based and coarse-grained.

Apta [33] examines a CXL environment where nodes can fail. It introduces a write-through coherence protocol for fault tolerance in Function-as-a-Service (FaaS) systems. It is a hardware-supported *transactional* protocol. Each transaction reads an object, operates on it, and uses special hardware to atomically push the updated object to the main memory node (there is a single main-memory node). It supports linearizability and, therefore, conflicting transactions are squashed. The protocol introduces lazy-invalidation, which moves invalidations out of the critical path of a write, so that the writer is not blocked if a sharer fails. Special scheduling support ensures lazy invalidations do not introduce inconsistencies.

In contrast to these three works, DIKTAMO targets general-purpose (not just transactional or application-specific) environments. Unlike Apta, it uses a write back protocol, does not require special scheduling, and works in a cluster with multiple memory nodes (not just one). Unlike all the schemes, it uses a novel hardware-supported, fine-grained recovery scheme for performance.

Resilience through replication and coherence protocols. Replication has long been used to achieve both resilience and performance in distributed and shared-memory systems. Works such as MINOS [34], FARM [15], PAR [1], and RAMCloud [31] replicate data across multiple nodes for high availability and low-latency access. Other work explores cache-line replication for fault tolerance in processor and memory systems. ICR [49] replicates active L1 lines by overwriting predicted-dead blocks. Replication Cache [48] extends this with a small replica cache storing dirty-line copies on each write. Fernandez-Pascual et al. [16] ensure at least one valid copy by delaying invalidations until safe transfer, tolerating interconnect faults. Dvé [32] replicates DRAM cache lines across NUMA sockets to provide resilience to memory errors, and maintains coherence among the replicas, but does not handle host-wide failures. In contrast, DIKTAMO logs updates in the context of CXL, enabling recovery from node crashes.

Shared-data reliability and coherence fault-tolerance has been studied for shared-memory multiprocessors [3], [8],

[16]. Our work complements these efforts by providing high-performance fault tolerance in distributed CXL systems.

IX. Conclusion

To address limitations of the CXL specification, this paper extends CXL to make it resilient to node failures. We call the system *DIKTAMO*. DIKTAMO augments the coherence transaction of writes to create replicas in the Logging Units of a few other nodes. Recovery after a node failure involves using the logs to bring the directory and memory to a correct state. Our evaluation with 16 4-core nodes shows that DIKTAMO enables fault-tolerant execution with 30% slowdown with three replicas (or 27% with two replicas) over the same platform with no fault-tolerance support. Further, DIKTAMO is 2.82x faster than ensuring reliability by using write-through writes.

Acknowledgment

This work was supported in part by NSF under grants CCF 2107470 and CCF 2316233; and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] R. Alagappan, A. Ganesan, E. Lee, A. Albarghouthi, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Protocol-Aware Recovery for Consensus-Based Storage," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/alagappan

[2] C. Alverti, S. Psomadakis, B. Ocalan, S. Jaiswal, T. Xu, and J. Torrellas, "CXLfork: Fast Remote Fork over CXL Fabrics," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds. ACM, 2025, pp. 210–226. [Online]. Available: https://doi.org/10.1145/3676641.3715988

[3] M. Banâtre, A. Gefflaut, P. Joubert, C. Morin, and P. A. Lee, "An Architecture for Tolerating Processor Failures in Shared Memory Multiprocessors," *IEEE Trans. Computers*, vol. 45, no. 10, pp. 1101–1115, 1996. [Online]. Available: https://doi.org/10.1109/12.543705

[4] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The Data Center as a Computer - Designing Warehouse-Scale Machines, Fourth Edition*. Springer, 2026. [Online]. Available: https://doi.org/10.1007/978-3-031-99489-0

[5] C. Bienia, S. Kumar, and K. Li, "PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors," in *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*, D. Christie, A. Lee, O. Mutlu, and B. G. Zorn, Eds. IEEE Computer Society, 2008, pp. 47–56. [Online]. Available: https://doi.org/10.1109/IISWC.2008.4636090

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *17th International Conference on Parallel Architectures and Compilation Techniques, PACT 2008, Toronto, Ontario, Canada, October 25-29, 2008*, A. Moshovos, D. Tarditi, and K. Olukotun, Eds. ACM, 2008, pp. 72–81. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst*, vol. 26, no. 2, pp. 4:1–4:26, 2008. [Online]. Available: https://doi.org/10.1145/1365815.1365816

[8] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault Containment for Shared-Memory Multiprocessors," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, M. B. Jones, Ed. ACM, 1995, pp. 12–25. [Online]. Available: https://doi.org/10.1145/224056.224059

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[10] CXL Consortium, "Compute Express Link (CXL) 3.2 Specification," https://computeexpresslink.org/wp-content/uploads/2024/11/CXL-Specification_rev3p2_ver1p0_2024October2_evalcopy.pdf, 2024, accessed: 2025-11-05.

[11] D. Das Sharma, "Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy," *IEEE Micro*, vol. 43, no. 2, pp. 99–109, 2023. [Online]. Available: https://doi.org/10.1109/MM.2022.3228561

[12] D. Das Sharma, R. Blankenship, and D. S. Berger, "An Introduction to the Compute Express Link (CXL) Interconnect," *ACM Comput. Surv*, vol. 56, no. 11, pp. 290:1–290:37, 2024. [Online]. Available: https://doi.org/10.1145/3669900

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, T. C. Bressoud and M. F. Kaashoek, Eds. ACM, 2007, pp. 205–220. [Online]. Available: https://doi.org/10.1145/1294261.1294281

[14] P. Deutsch, "GZIP file format specification version 4.3," *RFC*, vol. 1952, pp. 1–12, 1996. [Online]. Available: https://doi.org/10.17487/RFC1952

[15] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, R. Mahajan and I. Stoica, Eds. USENIX Association, 2014, pp. 401–414. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[16] R. Fernández-Pascual, J. M. García, M. E. Acacio, and J. Duato, "A Low Overhead Fault Tolerant Coherence Protocol for CMP Architectures," in *13th International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA*. IEEE Computer Society, 2007, pp. 157–168. [Online]. Available: https://doi.org/10.1109/HPCA.2007.346194

[17] S. Gurumurthi, A. J. Branover, B. Hornung, V. Michna, M. Natu, and C. Petersen, "An Overview of Reliability, Availability, and Serviceability (RAS) in Compute Express Link™ 2.0," CXL Consortium / Compute Express Link ™, White Paper, Jan 2021, accessed: 2025-11-08. [Online]. Available: https://computeexpresslink.org/wp-content/uploads/2023/12/CXL-RAS-Whitepaper-Post-WG-Revision_FINAL.pdf

[18] A. Hasbun and D. Walton, "OCP RAS API Revision 0.8," Open Compute Project, Tech. Rep., Feb 2023, accessed: 2025-11-08. [Online]. Available: https://www.opencompute.org/documents/2022-open-compute-specification-ras-api-v0-8-docx-pdf

[19] J. Huang, M. Zhang, T. Ma, Z. Liu, S. Lin, K. Chen, J. Jiang, X. Liao, Y. Shan, N. Zhang, M. Lu, T. Ma, H. Gong, and Y. Wu, "TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, E. Witchel, C. J. Rossbach, A. C. Arpaci-Dusseau, and K. Keeton, Eds. ACM, 2024, pp. 421–437. [Online]. Available: https://doi.org/10.1145/3694715.3695967

[20] Y. Huang, H. Chen, N. Ni, Y. Sun, V. Chidambaram, D. Tang, and E. Witchel, "Tigon: A Distributed Database for a CXL Pod," in *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, L. Zhou and Y. Zhou, Eds. USENIX Association, 2025, pp. 109–128. [Online]. Available: https://www.usenix.org/conference/osdi25/presentation/huang-yibo

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. C. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*, P. Barham and T. Roscoe, Eds. USENIX Association, 2010. [Online]. Available: https://www.usenix.org/conference/usenix-atc-10/zookeeper-wait-free-coordination-internet-scale-systems

[22] Intel, "Pin - A Dynamic Binary Instrumentation Tool," 2024. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[23] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search," in *Proceedings of the 2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, J. Lawall and D. Williams, Eds. USENIX Association, 2023, pp. 585–600. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/jang

[24] A. Katsarakis, V. Gavrielatos, M. R. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, "Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 201–217. [Online]. Available: https://doi.org/10.1145/3373376.3378496

[25] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "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, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, N. D. E. Jerger, and M. M. Swift, Eds. ACM, 2023, pp. 574–587. [Online]. Available: https://doi.org/10.1145/3575693.3578835

[26] J. Liu, H. Hadian, Y. Wang, D. S. Berger, M. Nguyen, X. Jian, S. H. Noh, and H. Li, "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, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds. ACM, 2025, pp. 1203–1217. [Online]. Available: https://doi.org/10.1145/3676641.3715987

[27] T. Ma, Z. Liu, C. Wei, J. Huang, Y. Zhuo, H. Li, N. Zhang, Y. Guan, D. Niu, M. Zhang, and T. Ma, "HydraRPC: RPC in the CXL Era," in *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, S. Bagchi and Y. Zhang, Eds. USENIX Association, 2024, pp. 387–395. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/ma

[28] S. Mahar, E. Hajyjasini, S. Lee, Z. Zhang, M. Shen, and S. Swanson, "Telepathic Datacenters: Fast RPCs using Shared CXL Memory," *CoRR*, vol. abs/2408.11325, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.11325

[29] Marvell Technology, Inc., "Marvell Launches Next-generation CXL Switch, Enabling Memory Pooling to Break Through the AI "Memory Wall"." https://investor.marvell.com/news-events/press-releases/detail/1017/marvell-launches-next-generation-cxl-switch-enabling-memory-pooling-to-break-through-the-ai-memory-wall, Mar. 2026, accessed: 2026-07-23.

[30] Micron Corporation, "CXL Memory Expansion: A Closer Look on Actual Platform," 2024. [Online]. Available: https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansion-a-close-look-on-actual-platform.pdf

[31] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 29–41. [Online]. Available: https://doi.org/10.1145/2043556.2043560

[32] A. Patil, V. Nagarajan, R. Balasubramonian, and N. Oswald, "Dvé: Improving DRAM Reliability and Performance On-Demand via Coherent Replication," in *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*. IEEE, 2021, pp. 526–539. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00048

[33] A. Patil, V. Nagarajan, N. Nikoleris, and N. Oswald, "Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS," in *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*. IEEE, 2023, pp. 201–215. [Online]. Available: https://doi.org/10.1109/DSN58367.2023.00030

[34] A. Psistakis, F. Chaix, and J. Torrellas, "MINOS: Distributed Consistency and Persistency Protocol Implementation & Offloading to SmartNICs," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2024, Edinburgh, United Kingdom, March 2-6, 2024*. IEEE, 2024, pp. 1–17. [Online]. Available: https://doi.org/10.1109/HPCA57654.2024.00076

[35] J. Ren, "Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB," https://github.com/basicthinker/YCSB-C, 2014.

[36] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. D. Kersey, R. A. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. L. Jacob, "The Structural Simulation Toolkit," *SIGMETRICS Perform. Evaluation Rev*, vol. 38, no. 4, pp. 37–42, 2011. [Online]. Available: https://doi.org/10.1145/1964218.1964225

[37] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, 2010. [Online]. Available: https://doi.org/10.1145/1785414.1785443

[38] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, M. G. Khatib, X. He, and M. Factor, Eds. IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: https://doi.org/10.1109/MSST.2010.5496972

[39] S. Wang, B. Hindman, and I. Stoica, "In Reference to RPC: It's Time to Add Distributed Memory," in *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, S. Angel, B. Kasikci, and E. Kohler, Eds. ACM, 2021, pp. 191–198. [Online]. Available: https://doi.org/10.1145/3458336.3465302

[40] Z. Wang, Y. Chen, C. Li, Y. Guan, D. Niu, T. Guan, Z. Du, X. Wei, and G. Sun, "CTXNL: A Software-Hardware Co-designed Solution for Efficient CXL-Based Transaction Processing," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, L. Eeckhout, G. Smaragdakis, K. Liang, A. Sampson, M. A. Kim, and C. J. Rossbach, Eds. ACM, 2025, pp. 192–209. [Online]. Available: https://doi.org/10.1145/3676641.3716244

[41] D. L. Weaver and T. Germond, *The SPARC Architecture Manual: Version 9*. Prentice Hall, 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:58276639

[42] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*, D. A. Patterson, Ed. ACM, 1995, pp. 24–36. [Online]. Available: https://doi.org/10.1145/223982.223990

[43] XConn Technologies, "XC50256 CXL 2.0 Switch Chip," Product datasheet, 2024. [Online]. Available: https://www.xconn-tech.com/product

[44] Y. Xu, S. Mahar, Z. Liu, M. Shen, and S. Swanson, "Position: CXL Shared Memory Programming: Barely Distributed and Almost Persistent," *CoRR*, vol. abs/2405.19626, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2405.19626

[45] X. Yang, Y. Zhang, H. Chen, F. Li, G. Fan, Y. Kong, B. Wang, J. Fang, Y. Wang, T. Huang, W. Hu, J. Kao, and J. Jiang, "Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases," in *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, V. Markl, J. M. Hellerstein, and A. Abouzied, Eds. ACM, 2025, pp. 689–702. [Online]. Available: https://doi.org/10.1145/3722212.3724460

[46] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu, "Adaptive Logging: Optimizing Logging and Recovery Costs in Distributed In-memory Databases," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1119–1134. [Online]. Available: https://doi.org/10.1145/2882903.2915208

[47] M. Zhang, T. Ma, J. Hua, Z. Liu, K. Chen, N. Ding, F. Du, J. Jiang, T. Ma, and Y. Wu, "Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory," in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, J. Flinn, M. I. Seltzer,

P. Druschel, A. Kaufmann, and J. Mace, Eds. ACM, 2023, pp. 658–674. [Online]. Available: https://doi.org/10.1145/3600006.3613135

[48] W. Zhang, "Replication Cache: A Small Fully Associative Cache to Improve Data Cache Reliability," *IEEE Trans. Computers*, vol. 54, no. 12, pp. 1547–1555, 2005. [Online]. Available: https://doi.org/10.1109/TC.2005.202

[49] W. Zhang, S. Gurumurthi, M. T. Kandemir, and A. Sivasubramaniam, "ICR: In-Cache Replication for Enhancing Data Cache Reliability," in *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*. IEEE Computer Society, 2003, pp. 291–300. [Online]. Available: https://doi.org/10.1109/DSN.2003.1209939

[50] W. Zheng, S. Tu, E. Kohler, and B. Liskov, "Fast Databases with Fast Durability and Recovery Through Multicore Parallelism," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, J. Flinn and H. Levy, Eds. USENIX Association, 2014, pp. 465–477. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_wenting

[51] Y. Zhong, D. S. Berger, C. A. Waldspurger, R. Wee, I. Agarwal, R. Agarwal, F. Hady, K. Kumar, M. D. Hill, M. Chowdhury, and A. Cidon, "Managing Memory Tiers with CXL in Virtualized Environments," in *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, A. Gavrilovska and D. B. Terry, Eds. USENIX Association, 2024, pp. 37–56. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/zhong-yuhong