CORD: Low-Latency, Bandwidth-Efficient and Scalable Release Consistency via Directory Ordering

Yanpeng Yu
Yale University
New Haven, CT, USA
yanpeng.yu@yale.edu

Nicolai Oswald
Nvidia
Santa Clara, CA, USA
noswald@nvidia.com

Anurag Khandelwal
Yale University
New Haven, CT, USA
anurag.khandelwal@yale.edu

Abstract

Increasingly, multi-processing unit (PU) systems (*e.g.*, CPU-GPU, multi-CPU, multi-GPU, etc.) are embracing cache-coherent shared memory to facilitate inter-PU communication. The coherence protocols in these systems support write-through accesses that place the data directly at the LLC to enable efficient producer-consumer communications pervasive in AI/ML workloads. Moreover, release consistency has emerged as the standard memory model in such systems due to its programming simplicity and ability to support high performance. In today's multi-PU systems, the source processor that issues the writes also orders them to enforce release consistency, even for write-through accesses. Unfortunately, such *source ordering* of write-through operations results in unnecessary communications between the source processor and the LLC directory, incurring significant performance, interconnect traffic, and energy overheads for multi-PU applications.

To eliminate such communication, we present CORD[1], a novel cache coherence protocol that orders write-through accesses directly at the *cache directory*. CORD employs several novel mechanisms to minimize the metadata required for ordering traffic while efficiently scaling to multiple directories. Evaluations atop the gem5 simulator show that compared to source ordering, CORD improves application performance by 24% and reduces traffic by 13% on average while incurring < 1% storage, area, and power overheads. Compared to hand-optimized message-passing implementations, CORD observes a mere 3% performance overhead and 6% more traffic on average with a significantly simpler programming model.

CCS Concepts

• **Computer systems organization** → **Multicore architectures**; **Interconnection architectures**; **Heterogeneous (hybrid) systems**.

Keywords

cache coherence protocol, memory consistency model, heterogeneous architecture

[1]**C**onsistency **OR**dered at **D**irectory

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731074

ACM Reference Format:
Yanpeng Yu, Nicolai Oswald, and Anurag Khandelwal. 2025. CORD: Low-Latency, Bandwidth-Efficient and Scalable Release Consistency via Directory Ordering. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3695053.3731074

1 Introduction

The slowdown of Moore's Law and the end of Dennard Scaling have driven the rise of a rich ecosystem of heterogeneous processing units (PUs). Recent architectures [20, 21, 24, 63] take this one step further, arguing for tight coupling of multiple PUs (*e.g.*, CPUs and GPUs, multiple CPUs or multiple GPUs) to facilitate low-latency, high-bandwidth inter-PU communication for emerging big-data workloads such as AI/ML [34, 35, 40, 52, 69, 71] and high-performance computing [17, 41, 43, 59, 62, 67]. Such architectures are increasingly adopting cache-coherent interconnects across PUs, such as NVIDIA's NVLink-C2C [23], AMD's Infinity Fabric [63], ARM's Advanced Microcontroller Bus Architecture Coherent Hub Interface (AMBA CHI) [1], and the Compute eXpress Link (CXL) [24]. These cache-coherent interconnects improve performance for a wide range of inter-PU use-cases [21, 45, 60, 70] with minimal programming complexity due to programmer-familiar memory models [28, 32, 42] — in stark contrast to increasingly complex message-passing mechanisms [2, 22] (§3.2).

While multi-PU shared memory is an active area of research, most approaches have converged on two design aspects (§2). First, most multi-PU cache coherence protocols [1, 9, 24] have added support for write-through or write-combining cache policies in addition to the traditional write-back policy due to their benefits to producer-consumer communication patterns between PUs, seen in emerging big data workloads such as AI/ML training and inference. Second, release consistency has emerged as the standard multi-PU shared memory model due to its programming simplicity and performance. For instance, HSA [6, 32] and the NVIDIA PTX [42] memory models employ release consistency semantics. We defer the details of release consistency semantics via Acquire, Release, Acquire-Release, and Relaxed annotations to §2.2.

In today's multi-PU systems, release consistency is enforced at the source processor, i.e., the processor issuing the memory operation — even for write-through accesses committed at the last-level cache (LLC). For instance, the source processor pipeline must not issue a Release store until the cache directory acknowledges all prior write-through accesses in program order using the underlying cache coherence protocol. Concrete examples include ARM's AMBA CHI specification [1] and the CXL 3.0 specification [24], both of which delegate the enforcement of write-through access ordering to the source processor via acknowledgment messages (Fig. 1 (left)).

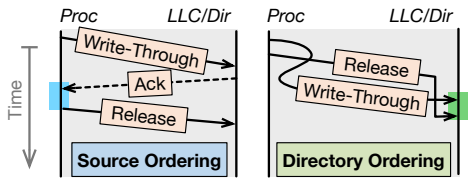

Figure 1: Unlike source ordering (left), CORD (right) orders write-through stores directly at the cache directory, eliminating inefficiencies stemming from acknowledgments.

We refer to this mechanism of enforcing release consistency at the source processor as *source ordering*.

Interestingly, we find that the need for acknowledgment messages for write-through accesses is not fundamental. Indeed, this requirement is tied to the source-ordering approach of ordering all memory operations at the source processor, even though write-through accesses must be committed at the LLC directory. The additional acknowledgment messages incur significant interconnect latency, traffic, and energy overheads for multi-PU applications. Message-passing protocols such as PCIe 5.0 can facilitate write-through communication with a release consistency-like programming model while avoiding unnecessary acknowledgment messages since they order messages at the *destination* rather than the source. However, the destination-ordering mechanism of message passing can only enforce point-to-point release consistency rather than enforcing such guarantees system-wide. As such, programs written using message passing in multi-PU systems incur significant programmer complexity for implementing release consistency at an application level. The shortcomings of source-ordering and message-passing approaches are detailed in §3.

To simultaneously achieve programming simplicity and efficiency, we present CORD (**C**onsistency **OR**dered at **D**irectory), a novel cache coherence protocol that, as the name suggests, orders write-through accesses at the cache directory — ensuring that the ordering and commitment of such requests occur at the same location. This, in turn, avoids the need for superfluous acknowledgment messages between the source processor and the directory (Fig. 1 (right)).

Realizing directory ordering in CORD requires resolving several unique challenges (§4). First, while adding sequence numbers to write-through requests offers a natural means for ordering such requests at the directory, the number of bits allocated to these sequence numbers introduces interesting trade-offs between the traffic overhead due to too many added bits in requests and the overheads due to overflow handling when the bits are too few. To this end, CORD *decouples* sequence numbers into coarse-grained epoch numbers and fine-grained store counters — tailored for release consistency in a way that strikes a balance between performance and traffic overheads (§4.1).

Second, while ordering requests at the directory eliminates unnecessary acknowledgment messages, scaling this approach to a multi-directory system requires careful treatment to preserve release consistency while ensuring performance scaling. To this end, CORD employs a novel inter-directory notification mechanism, where directories directly notify each other to signal the completion of certain store operations, enforcing cross-directory ordering without involving the processor. This approach not only avoids processor stalls but also minimizes write-through latency and interconnect bandwidth usage (§4.2).

Finally, the techniques required to ensure low latency and bandwidth efficiency in CORD also require additional storage at the various processors and directories in a multi-PU system. Left unchecked, this added storage can incur significant area and power overheads and ultimately render CORD impractical. To address this, we employ a practical resource provisioning technique to upper-bound the storage overhead for CORD (§4.3) without degrading application performance.

We have verified CORD's correctness using the Murphi [26] model checker with both classic and customized litmus tests (§4.5). Evaluations using the gem5 [14] simulator (§5) show that compared to the state-of-the-art multi-PU source-ordered cache coherence protocol, Spandex [9], CORD improves end-to-end application performance by 24%, reduces the interconnect traffic by 13% while incurring < 1% storage, area, and power overheads. Compared to hand-optimized message-passing implementations, CORD observes a mere 3% performance and 6% traffic overhead on average with a significantly simpler programming model.

In summary, this paper makes the following contributions:

- We demonstrate the performance and bandwidth inefficiencies of source ordering and how the point-to-point message-passing model can violate release consistency in multi-PU systems.
- We propose CORD, a novel cache coherence protocol that orders write-through accesses efficiently at the cache directory using decoupled sequence numbers, inter-directory notifications, and bounded storage provisioning.
- We establish CORD's correctness using the Murphi model checker and demonstrate its benefits (and overheads) over source ordering and message passing using the gem5 simulator for a wide range of real-world and synthetic workloads.

2 Background

2.1 Inter-PU Write-Through Cache-Coherence

Recent multi-PU systems [20, 21, 23, 24, 63] are increasingly embracing cache-coherent shared memory to facilitate inter-PU communication. For example, NVIDIA's Grace Hopper Superchip [21, 23] and AMD's Accelerated Processing Units (APU) [63] coherently interconnects CPU and GPU chips on the same motherboard, while the Compute eXpress Link (CXL) [24] envisions cache-coherent shared memory across diverse heterogeneous computing devices at rack-scale or beyond. The broad adoption of cache-coherent shared memory stems from the flexibility it enables for various multi-PU use cases, such as CPU-GPU collaborative computing [45], CPU-CPU shared-everything computing [70], fine-grained GPU memory oversubscription [21], and CPU-NIC memory sharing [60]. Shared memory also reduces programming complexity compared to message-passing mechanisms atop PCIe [2] and NVLink [22] since it frees programmers from orchestrating low-level data movement across PUs by hiding them under cache coherence protocols.

While traditional CPU cache coherence protocols [4, 10, 27, 29, 53] have mainly employed the write-back policy — where stores

are flushed to next-level cache only on eviction to exploit locality — most inter-PU cache coherence protocols [1, 9, 24] have additionally employed the write-through (or write-combining) policy, where stores or atomics are *directly* propagated to the last-level cache or main memory. For example, the AMBA CHI specification [1] defines the `WriteUnique` or `WriteNoSnp` store types for write-through stores while CXL 3.0 [24]'s CXL.io defines Unordered IO (UIO) write for the same use.

The popularity of the write-through policy for multi-PU cache coherence stems from its benefits for various emerging multi-PU workloads, such as machine learning training and inference, or other pipeline workloads. Specifically, these workloads often employ producer-consumer communication between CPU and GPU or between GPUs, where the write-through policy allows the producer PU to directly propagate updates to the consumer PU's memory hierarchy with minimal latency and traffic.

2.2 Release Consistency

While memory models across multiple PUs, such as Nvidia PTX [42] and the Heterogeneous System Architecture (HSA) memory model [6, 32], are an active area of research, most approaches have converged to provide release consistency or its variants [6, 28, 32, 42]. Release consistency enjoys a sweet spot between programming simplicity and performance. It is supported as a software memory model in various modern languages, including C++, Rust, Java, and OpenCL. This has driven its broad adoption in multi-PU memory models.

While we refer the reader to [12] for one of many formal definitions of release consistency, we provide an informal description of its requirements here. Intuitively, release consistency allows memory accesses or fences to be annotated with Acquire, Release, Acquire-Release, or Relaxed labels. Stores, atomics, or fences annotated with Release serve as barriers that prior memory accesses (in program order) cannot be reordered after. Similarly, loads, atomics, or fences annotated with Acquire serve as barriers that subsequent memory accesses (in program order) cannot be ordered before. Finally, while Acquire-Release accesses serve as a full memory barrier (i.e., no accesses can be reordered before or after them), Relaxed accesses do not have any ordering constraints.

3 Motivation

We motivate the need for CORD by highlighting the shortcomings of source ordering (§3.1) and message passing (§3.2) for achieving release consistency.

3.1 Inefficiencies of Source Ordering

In today's multi-PU systems, release consistency is enforced at the source processor even for write-through stores or atomics that are completed at the last-level cache (LLC). For instance, as per the Release semantics introduced in §2.2, the source processor must not issue a Release until the cache-coherence directory has acknowledged the completion of all prior write-through accesses in program order. For example, the AMBA CHI specification [1] refers to the ordering between write-through stores as Ordered Write Observation (OWO), which is enforced by having the cache directory issue an acknowledge message back to the source processor for each write-through access (*e.g.*, `WriteUnique`, `WriteNoSnp` or atomics). Similarly, while only supporting IO-coherence, CXL 3.0 [24]'s CXL.io protocol uses Unordered IO (UIO) write transaction to offload the enforcement of write ordering completely to the source processor using the UIO write completion message. We refer to this source-based mechanism of enforcing release consistency for write-through coherence as *source ordering*.

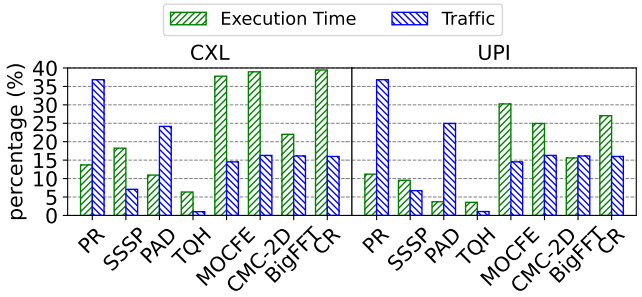

Figure 2: Source ordering's acknowledgments incur significant performance and traffic overheads (§3.1).

Unfortunately, a key source of inefficiency in source ordering is its requirement of ordering write-through accesses at the source processor, while the completion of these operations is delegated to the destination cache directory. Specifically, ordering and completing these accesses at two locations results in significant performance, traffic, and energy overhead due to the additional acknowledgment messages from cache directories to processors. Such acknowledgments not only delay Release stores by at least one interconnect round-trip — degrading performance — but also incur interconnect traffic (and corresponding energy consumption) proportional to the communicated data size. To understand how such inefficiencies affect real-world applications, we broke down source ordering's execution time spent waiting for write-through acknowledgments and the interconnect traffic generated by acknowledgments for our evaluated applications (details of the applications are deferred to §5). We evaluate these overheads using simulated CXL [24] and Intel UPI [33] as the baseline inter-PU interconnects for the system.

Fig. 2 shows that significant performance and traffic overhead are observed across all applications. Specifically, for CXL, except `TQH`, all other applications spend over 10% execution time waiting for acknowledgments, while except `SSSP` and `TQH` all observe over 14% traffic overhead. `CMC-2D`, `MOCFE`, and `CR` observes even more than 37% slowdown, while `PR` observes over 36% traffic overhead. Even though UPI incurs much lower latency, applications still observe 4% – 30% slowdown and 1% – 30% traffic overhead. We conduct a more in-depth analysis of these overheads in §5.2.

3.2 Message Passing Does Not Provide Release Consistency

To avoid the performance and traffic overheads incurred by source ordering's acknowledgments, message-passing protocols such as PCIe 5.0 [2] are an alternative mechanism to facilitate inter-PU write-through communication with a release consistency-like programming model. For example, PCIe allows a PU to write through messages to another PU by using "posted" PCIe-write transactions

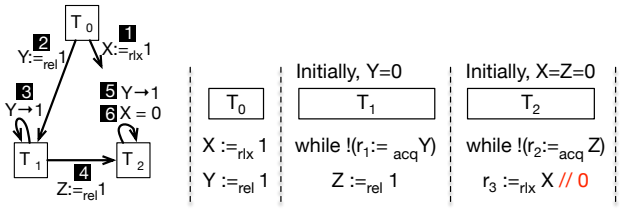

Figure 3: Message passing allows execution outcome forbidden by release consistency for a variant of the ISA2 litmus test (§3.2). The litmus test program is shown on the right, and the message-passing execution diagram on the left.

to modify the latter's memory hierarchy directly, without any acknowledgments. This is because the ordering is directly enforced at the destination endpoint. PCIe even defines *Relaxed Ordering* and *Strong Ordering* for write transactions, whose ordering rules are similar to shared memory's Relaxed and Release stores. However, such ordering guarantees are restricted to point-to-point communications and not enforced across all PUs in the system, breaking release consistency for multi-PU systems.

Indeed, we show that message passing allows outcomes forbidden by release consistency for a variant of the classic ISA2 litmus test [58]. The ISA2 litmus test shown in Fig. 3 (right) employs three threads T_0, T_1, and T_2, and three variables (X, Y, Z, all initially zero) where X and Z are mapped to T_2's memory, while Y is mapped to T_1's memory. Fig. 3 (left) shows a possible execution of the text with message passing, where the three threads pass messages in a pipelined manner: T_0 first remotely sets X in T_2's memory (1), then remotely sets Y in T_1's memory (2). T_1 first locally polls Y in its own memory (3), then remotely sets Z in T_2's memory (4). T_2 first locally polls Z in its own memory (5), then locally loads X in its own memory (6). The message-passing execution in Fig. 3 (left) shows that even though:

- T_1's polling on Y 3 must wait for T_0's message 2, and,
- T_2's polling on Z 5 must wait for T_1's message 4,
- T_2's load 6 need not wait for T_0's message 1 since T_0 and T_2 do not employ point-to-point *Acquire*-Release synchronization, i.e., T_2's $r_3 :=_{rlx} X$ can return 0.

However, release consistency forbids such an outcome because T_0 and T_2 perform indirect synchronization through T_1 using Y and Z (i.e., it enforces *synchronization cumulativity* [58]).

A more practical impact of this shortcoming was observed in one of our evaluated workloads, TQH, from the Chai benchmark [30], which encounters an error pattern similar to ISA2 with message passing. As such, we could not even evaluate its performance and traffic under message passing in §5.2.

Since naive message passing in a multi-PU system breaks release consistency, careful use of message-passing semantics is required for correctness. Specifically, programmers must explicitly orchestrate point-to-point communications in the right order to preserve release consistency, incurring high programming complexity. Moreover, this complexity will only increase as multi-PU systems embrace increasingly complex interconnect topologies [25].

This raises the research question: Can we achieve message-passing-like efficiency while maintaining the programmer-familiar release consistency model? We answer the question in the affirmative by presenting CORD, a *directory-ordered* write-through cache coherence protocol, as detailed in the next section.

4 CORD Design

Unlike source ordering, which orders stores at the source processor using acknowledgments to enforce release consistency, CORD directly orders stores at the destination cache directory.

We begin by detailing how write-through accesses are ordered at the directory for a single directory system, leveraging epoch numbers and store counters to maximize performance and minimize traffic overheads (§4.1). We then describe how CORD's novel inter-directory notification mechanism enables scalable release consistency across multiple directories while minimizing latency and communication traffic compared to source ordering — the de facto mechanism for enabling release consistency across multiple directories (§4.2). Next, we describe how we provision storage resources to CORD data structures to limit its storage overheads in §4.3. Finally, we describe how we model check its correctness in §4.5.

4.1 Ordering for a Single-directory System

We begin by considering a simple system with a single shared directory. With a naive approach, ordering write-through stores at the directory in a single-directory system requires (*i*) tracking a sequence number for each request at the source and embedding the sequence number in the request and (*ii*) committing the stores at the directory according to their sequence numbers and consistency semantic. However, since sequence numbers are fixed-width, once they reach their limit and are about to overflow, a processor must stall until all prior sequence numbers are ordered and the sequence number can be reset. As such, the bit width of the sequence number exposes a trade-off: small bit-widths incur performance degradation stemming from frequent overflow handling, while large bit-widths incur high traffic overheads by inflating the request sizes.

CORD breaks this trade-off by decoupling sequence numbers with *epochs* and *store counters*. At a high level, epochs divide logical time into periods between Release stores, while store counters track sequence numbers for Release stores between epochs and are reset at every new epoch. Further, CORD embeds the complete sequence number (i.e., epoch number and store counter) only in the infrequent Release store requests, while embedding only the epoch number in the more frequent Relaxed store requests. By employing small bit-width epoch numbers that do not inflate Relaxed stores and large bit-width store counters that are infrequently overflowed, CORD can achieve the best of both worlds.

We next illustrate how CORD's epoch numbers and store counters order write-through stores at the directory, then demonstrate how our approach breaks the trade-off between performance degradation and traffic overhead. For demonstration simplicity, we break down our illustrations along two dimensions: (*i*) enforcing ordering between Relaxed and Release stores and (*ii*) enforcing ordering between different Release stores. Together, they are necessary and sufficient to ensure write-through accesses follow release consistency.

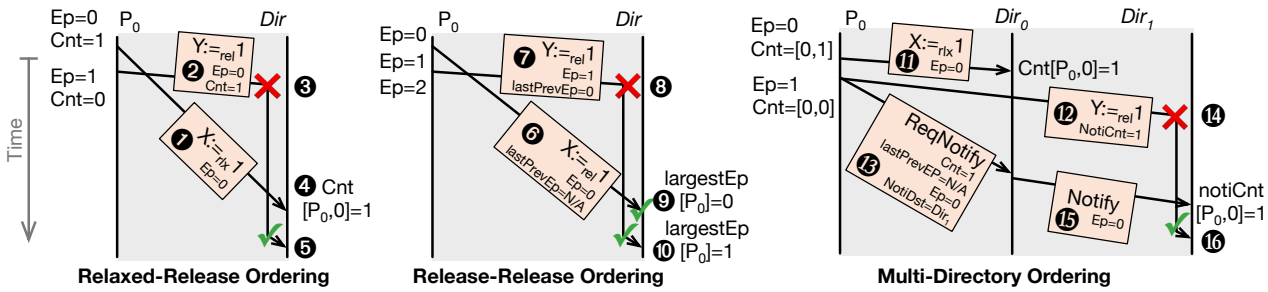

Figure 4: Examples of CORD's epoch number and store counter (§4.2) for relaxed-release ordering (left) and release-release ordering (middle), and CORD's inter-directory ordering for multi-directory release consistency (right, §4.2).

Enforcing Relaxed-Release **Ordering.** Correctness under release consistency requires preventing reorders between a prior Relaxed store and a subsequent Release store. To ensure this, each processor core in CORD locally maintains an epoch number (Ep) that is incremented on each Release store and a store counter (Cnt) that is incremented on each Relaxed store and reset on each Release store. Intuitively, each epoch number corresponds to a Release store, and the store counter tracks the number of Relaxed stores starting from the last Release store. Each directory also maintains a mapping from processor core ID and epoch number to store counters (Cnt[PID, Ep]), which track the number of Relaxed stores committed at this directory for a specific processor-epoch pair.

Fig. 4 (left) shows a minimal example of how CORD enforces the ordering between a Relaxed store and a subsequent Release store hosted by the same directory. Specifically, the processor embeds only the epoch number in each Relaxed store request (❶) while embedding both the epoch number and store counter in each Release store request (❷). When a Relaxed store arrives at the directory, the directory immediately commits it and increments its store counter for the corresponding processor-epoch pair (❹). In contrast, when a Release store arrives at the directory, it can only commit the store request if its embedded store counter matches the store counter the directory maintains for the corresponding processor and epoch (❺). Otherwise, the Release store is stalled (❸). This ensures that a Relaxed store is always committed before a subsequent Release store if the same directory handles both.

Enforcing Release-Release **Ordering.** Fig. 4 (middle) shows a minimal example of how CORD enforces the ordering between two Release stores at the same directory. The store counters mentioned above are omitted for demonstration simplicity. Specifically, within each Release store request, the processor embeds the last prior epoch number for which the corresponding release store has been issued to the destination directory but has not been acknowledged (lastPrevEp; ❻ and ❼; note that CORD still requires Release stores to be acknowledged). The directory needs to track, for each processor, the largest committed epoch (largestEp[PID]; ❾ and ❿). It can commit a Release store only if its embedded last prior epoch has been committed (❾ and ❿). Otherwise, the Release store is stalled (❽). This ensures that Release stores are committed following their program order.

Balancing bandwidth and overflow handling overheads. Since only the epoch number is embedded in the majority of the traffic (i.e., Relaxed stores), CORD can use low-bit-width representations to reduce traffic overheads. For instance, our implementation of CORD atop CXL 3.0 adopts an 8-bit epoch number, which can entirely fit in CXL 3.0 transaction packets' reserved bits, incurring no traffic overheads for Relaxed stores. Meanwhile, small epoch numbers do not cause performance degradation due to frequent overflow handling since epoch numbers are incremented only on Release stores, which typically span a few to tens of kilobytes of Relaxed data stores as shown by our evaluated workloads (§5.2). Overflows for store counters are even more infrequent, as CORD can adopt sufficiently large store counters given that it only adds overhead for the infrequent Release store messages. For example, our implementation employs a 32bit store counter that can support 32GB of 8B stores without overflow while incurring only 4B traffic per each Release store, which typically spans at least several kilobytes of Relaxed stores in our evaluated workloads. We empirically show the benefits of CORD's decoupled epoch numbers and store counters in §5.3.

4.2 Ordering across Multiple Directories

As most multi-PU systems comprise more than one directory, the directory-ordering mechanism outlined above must be extended to multiple directories to enable scalable release consistency semantics in a multi-directory system. CORD adopts a novel inter-directory notification mechanism to achieve this goal with reduced latency and traffic compared to source ordering. At a high level, with inter-directory notification, the directories directly notify each other to signal the completion of certain stores, enforcing cross-directory ordering without involving the source processor.

Next, we detail the inter-directory notification mechanism and discuss why it improves performance and reduces traffic compared to source ordering.

Inter-directory notification. Fig. 4 (right) shows a minimal example of how inter-directory notification enables release-consistent ordering between a Relaxed and a subsequent Release store at two different directories. Enforcing Release-Release ordering is similar. Specifically, when issuing a Release store to a destination directory, in addition to the epoch number, store counter, and the last prior

unacknowledged epoch number (as outlined above), the processor also embeds a notification counter (NotiCnt; ⑫), which tracks the number of other directories (referred as *pending* directories) that either has one or more pending Relaxed store(s) in the current epoch or has one or more unacknowledged Release store(s). This notification counter indicates to the destination directory that the current Release store should not be committed before notifications from all pending directories are received.

Concurrently, the processor sends a "request for notification" message (⑬) to each pending directory. This request contains (*i*) the number of Relaxed stores in the current epoch for the target pending directory, (*ii*) the last unacknowledged epoch for the target pending directory, (*iii*) the current epoch number, and (*iv*) destination directory of the current Release store (NotiDst). Intuitively, this message informs the target pending directory about all pending Relaxed/Release stores up to the current epoch and requests it to notify the destination directory after it commits all pending stores.

Upon receiving a request for notification, a pending directory sends a notification to the destination directory (⑮) after it commits all the pending stores as embedded. The destination directory can commit a Release store only after its collected number of notifications for the corresponding processor and epoch number (notiCnt[PID, Ep]) equals the notification counter embedded in the Release store message (⑯). This ensures that a Release store will not be committed until all prior stores at other directories have been committed, ensuring release consistency.

Improving performance and reducing traffic. Compared to source ordering, CORD's inter-directory notification consistently improves performance (i.e., interconnect round-trips) while reducing traffic under most scenarios as we evaluated in §5.3. We use an intuitive example to illustrate these benefits (Fig. 5), where a processor issues m Relaxed stores in total to the first $n-1$ directories (i.e., $\text{Dir}_0 ... \text{Dir}_{n-2}$), followed by a Release store to the last directory (i.e., Dir_{n-1}).

With source ordering (Fig. 5 left), the processor stalls 2 interconnect hops to wait for Relaxed store acknowledgments while the Release store takes 3 interconnect hops to reach the directory. Source ordering also generates $m+1$ control messages consisting of all the acknowledgments.

In contrast, with CORD's inter-directory notification (Fig. 5 right), the source processor does not stall as it does not wait for any acknowledgments. The Release store takes at most two interconnect hops to reach all directories — one hop for the request for notification to be received by all pending directories and one hop for the actual notification message from the pending directories. As such, CORD enjoys lower processor stall time *and* store latency compared to source ordering. In the worst case, CORD generates $2n-1$ control messages, i.e., when all directories are pending directories, requiring $n-1$ requests for notification, $n-1$ notifications, and 1 acknowledgment. While this can potentially exceed the number of messages in source ordering (depending on the workload), we find that optimized real-world multi-PU applications typically restrict their communication fan-out, i.e., a small number of pending directories, resulting in smaller effective n. Moreover, real-world applications employ inter-PU synchronization granularities of at least a few kilobytes (i.e., large m) to minimize communication overheads. As such, inter-directory notifications' traffic overhead is smaller than source ordering for most real-world applications (as we will show in §5.2).

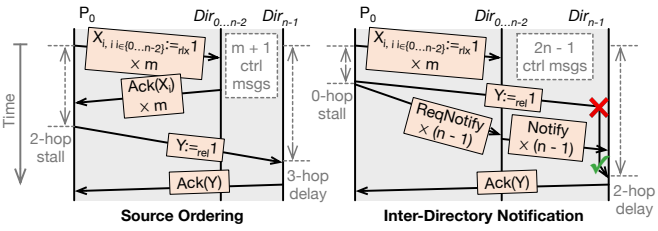

Figure 5: Compared to source ordering (left), CORD's inter-directory notification (right) eliminates processor stall while reducing write-through delay and traffic overheads for enforcing multi-directory release consistency.

Algorithm 1 CORD protocol at processor.

```
1:  procedure On Relaxed store
2:      Embed epoch number to Relaxed store                    ▷ §4.1
3:      Increment store counter
4:      Send Relaxed store to its destination directory

5:  procedure On Release store
6:      Embed epoch and store counter to Release store         ▷ §4.1
7:      Track the current epoch as unacknowledged
8:      Increment epoch and reset store counter
9:      Embed last unacknowledged epoch to Release store
10:     for all pending directories do                         ▷ §4.2
11:         Send request for notification to the pending directory
12:     Embed pending directory count to Release store
13:     Send Release store to its destination directory

14: procedure On Release store Ack
15:     Mark epoch acknowledged                                ▷ §4.1
```

Algorithm 2 CORD protocol at directory.

```
18: procedure On Relaxed store
19:     Commit Relaxed store to LLC                            ▷ §4.1
20:     Increment store counter

21: procedure On Release store
22:     if embedded store counter matches the directory's
        and last unAck-ed epoch committed                      ▷ §4.1
        and all inter-directory notifications received then    ▷ §4.2
23:         Commit Release store to LLC
24:     else Retry later

25: procedure On Request for Notification
26:     if embedded store counter matches the directory's
        and last unAck-ed epoch committed then                 ▷ §4.2
27:         Send notification to the destination directory
28:     else Retry later

29: procedure On Notification
30:     Increment notification count                           ▷ §4.2
```

Putting it all together. Algorithms 1 and 2 detail the CORD protocol at the processor and directory, respectively.

At the processor, each Relaxed store request is embedded with the epoch number and triggers the increment of the store counter (lines 2-3). Conversely, each Release store request is embedded with the epoch number, store counter, the last unacknowledged epoch number, and the pending directory count (lines 6,9,12). The Release store also triggers an increment of the epoch number, a reset of the store counter, and a potential inter-directory request for notification (lines 7, 8, 10-11).

At the directory, Relaxed stores can be immediately committed to LLC (line 19) while each Release store can be committed if and only if (*i*) its embedded store counter matches the one maintained by the directory, (*ii*) the source processor's last unacknowledged epoch has been committed, and (*iii*) all inter-directory notifications are collected (line 22). In addition, a request for notification triggers a notification to be sent to the destination directory when all Relaxed and Release stores are committed at the current pending directory (line 26).

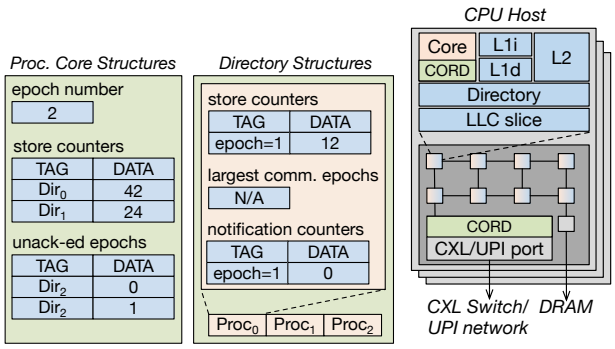

Figure 6: Micro-architectural implementation of our data structures at the processor core and directory (left, §4.3), and simulated system architecture (right, §5.1).

4.3 Bounding CORD's Storage Overhead

CORD introduces storage overheads due to the data structures that track protocol states at the processor core and the directory. While §4.1 and §4.2 described how these data structures are used, we now detail their micro-architectural implementations and how we bound their storage overheads.

Implementing data structures in CORD. As shown in Fig. 6 (left), each processor maintains three data structures: the current epoch number, the store counters for each directory in the current epoch, and the unacknowledged epoch numbers for each directory. All structures are implemented as look-up tables except for the epoch number, which is realized as a single counter. The directory also maintains three data structures: the store counter for each processor core and each of its epochs, the largest committed epochs for each processor core, and the notification counter for each processor core and each of its epochs. All structures in the directory are implemented per-processor-core with statically partitioned storage to handle look-up table overflow under worst-case scenarios, as described next.

Since several structures are maintained per epoch, storage for acknowledged or committed epochs can be reclaimed. Specifically, the processor removes an unacknowledged epoch entry after it is acknowledged (after line 15). The directory removes its store counter and notification counter entry after an epoch is committed (after line 23) and removes its store counter entry after the notification is sent for an epoch (after line 27). This ensures that the storage for CORD does not accumulate indefinitely. Moreover, as we show next, we also upper-bound CORD's consumed storage in practice.

Bounding storage overheads. While storage can be reclaimed for expired epochs, in the worst case, they still consume as many entries as the total number of epochs, potentially leading to large storage overheads. For example, assuming the epoch number is 8 bits, the store counter table at the directory would need to hold 256 entries in the worst case, i.e., to handle the case where 256 consecutive Release stores from the same processor arrive at the directory in complete reverse order. Assuming 64 processor cores in the system and 4-byte store counters, each directory would consume over 64KB storage, which is prohibitively large.

However, such worst-case scenarios are *extremely* rare — indeed, they did not occur for *any* of our evaluated workloads and systems (§5.2). For context, a complete reversal of order across 256 consecutive Release stores, even for a workload that issues Release store every 1*ns*, requires that the first and last Release store's transmission time over the interconnect differ by 256ns. Not only is the total interconnect latency for even higher latency interconnects like CXL 3.0 only 150ns [39], but no practical multi-PU workload would issue Release stores as frequently (see Table 2 for a characterization of popular workloads). Indeed, we validate this observation for both real-world workloads and synthetic worst-case benchmarks over diverse system configurations in §5.4.

As such, CORD provisions sufficient storage for practical scenarios while guaranteeing correctness for the worst-case scenario by *stalling*. Specifically, CORD processors stall Release stores if they detect that such a store will overflow any look-up tables. In more detail, before a processor issues a Release store, it (*i*) checks locally whether the unacknowledged epoch table has an available entry, and (*ii*) checks whether its destination directory's store counters and notification counters have an available entry, by comparing the storage statically allocated to the initiating processor and the storage upper-bound required by the total number of unacknowledged Release stores. The Release store is stalled if either check fails until sufficient space becomes available via the completion of pending Release stores.

4.4 Interactions with Other Memory Accesses

While CORD targets efficient ordering between write-through stores, it must also ensure the ordering between them and other memory accesses such as write-back stores, loads, barriers, and instructions with data, address, or control dependencies. We now detail how CORD enforces such ordering.

Write-back stores. CORD does not change ordering for write-back stores — they are still source-ordered with any (necessary) memory accesses. The only exception is enforcing ordering between an earlier directory-ordered Relaxed write-through store and a subsequent Release write-back store. This is because the former does not have an acknowledgment and thus cannot be source-ordered with the latter. In this case, the processor injects an additional directory-ordered Release barrier after the former and stalls until the barrier is acknowledged before it issues the latter.

Loads. CORD does not change ordering for loads — Relaxed loads are source-ordered with subsequent Release stores and preceding Acquire loads using acknowledgments, while Acquire loads are source-ordered with subsequent memory accesses.

Barriers. CORD requires additional handling for memory barriers to order write-through stores. We implemented three types of memory barriers with CORD: Acquire, Release, and sequentially-consistent. Specifically, in addition to ordering regular accesses such as write-back stores and loads, a Release or sequentially-consistent barrier requires the processor to broadcast an "empty" directory-ordered Release store to all pending directories and wait for their acknowledgments. An Acquire barrier, on the other hand, does not need additional handling beyond a regular Acquire barrier because it already ensures the completion of all preceding loads, which suffices for Acquire's semantics even in the presence of directory-ordered stores.

Dependencies. To enforce instruction dependencies (address, data, and control), we conservatively inject full memory barriers between dependent memory operations to ensure completion, without adding custom logic to CORD. A variety of fine-grained dependency enforcement techniques in out-of-order processors [55] can potentially improve CORD's performance; we leave their exploration for future work.

4.5 Correctness

We have model-checked CORD using the Murphi [26] model checker. Like many prior works on cache coherence validation [18, 48–50, 54], our model checker avoids state space explosion by limiting the validation for up to four addresses, three data values, and four nodes — each with a private cache and a directory.

Using Murphi, we run 122 Armv8 release consistency litmus tests generated using the herd tool [7] and additionally run 180 customized litmus tests. We use these customized litmus tests to cover various design spaces and scenarios that trigger potential corner cases. For example, our tests cover the scenario where only some processor cores in the system use CORD while other cores stick to the traditional source ordering, the case where a single processor core issues both directory-ordered and source-ordered write-through stores, the case where the processor or directory look-up table storage is under-provisioned, and the case where epoch numbers, store counters, or notification counters overflow. All of our litmus tests have passed, establishing CORD's ability to enforce release consistency safely, as well as its deadlock-freedom.

Processor	
# of cores per CPU host	8
# of CPU hosts	8
Cache Hierarchy	
Private L1I&D caches	64KB/core, 2-way, 2-cycle latency
Private L2 caches	64KB/core, 8-way, 4-cycle latency
Shared LLC cache	64 slices of 2MB, 8-way, 8-cycle latency
Interconnect	
Intra-host topology	2 × 4 mesh
Inter-host topology	Single switch
Intra-host link latency	10 Cycles
Inter-host link latency	150ns (CXL); 50ns (UPI)
link bandwidth	64GB/s bidirectional
Memory	
Size	HBM4, 4GB per host
Bandwidth	8 channels, 64GB/s per channel

Table 1: Simulation configuration for all schemes (§5.1).

Suite	**Name**	**Input**	Relaxed **Gran.**	Release **Gran.**	**Comm. Fanout**
Pannotia	PR	olesnik	word	5KB	High
	SSSP	wing	word	700B	High
Chai	PAD	basket	line	1KB	Medium
	TQH	1024*1024	line	8B-2KB	Low
	HSTI	basket	line	1KB	Medium
	TRNS	1024*1024	line	512B	High
DOE (MPI)	MOCFE	N/A	word/line	8B-256B	High
	CMC-2D	N/A	line	1B-14KB	High
	BigFFT	N/A	word/line	10KB	Low
	CR	N/A	line	8B-2KB	Low

Table 2: Evaluated benchmarks (§5.1).

5 Evaluation

In this section, we evaluate CORD to answer the following questions:

(1) To what extent does CORD improve performance and interconnect traffic for end-to-end workloads (§5.2)?
(2) What combination of workloads and system settings is CORD most and least useful for? (§5.3)
(3) What are CORD's storage, area, and power overheads? (§5.4)

5.1 Methodology

Simulated architecture. Our simulated multi-PU system models the hardware-based coherent memory realization across multiple hosts described in the CXL 3.0 specification [24] as shown in Fig. 6 (right). It comprises multiple CPU hosts connected via a CXL switch. Each CPU host consists of a mesh of cores. Each core employs a core-private L1 data and instruction cache, a core-private L2 cache, and a slice of shared Last-level cache (LLC) co-located with its cache directory. The simulated CXL multi-CPU system parameters are summarized in Table 1. Our simulation models CXL performance as reported in the recent study from Microsoft [39], where the round-trip latency between two hosts is an optimistic ~ 150 ns. Since CORD's benefits are more pronounced at larger interconnect latencies, this reflects a lower bound on its benefits. To

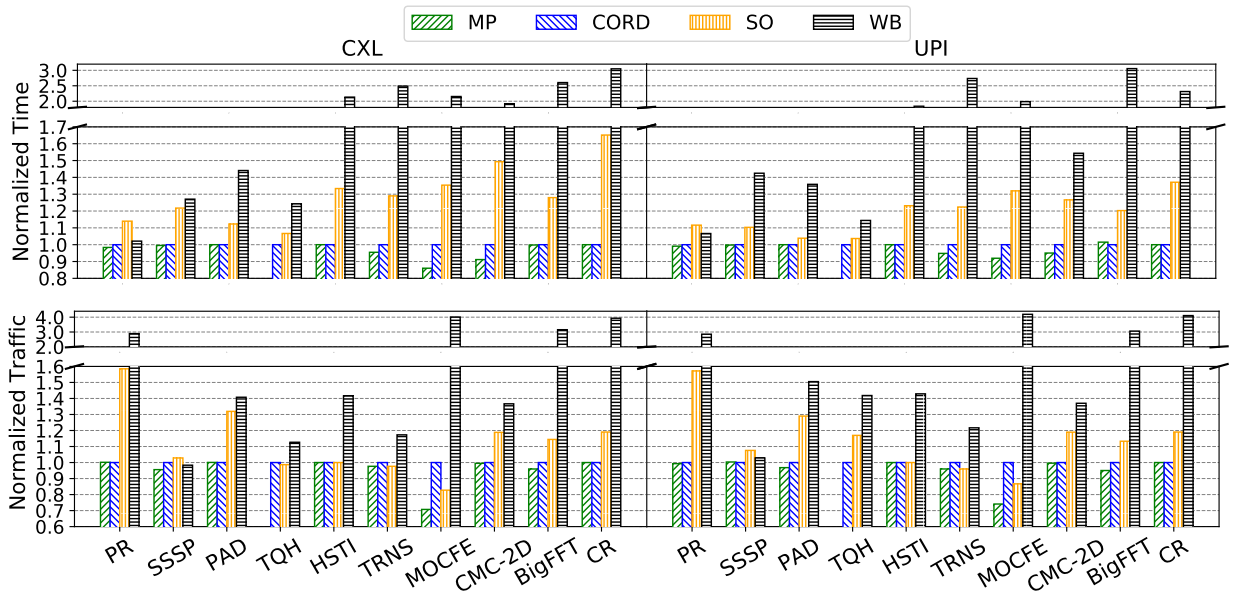

Figure 7: Performance (top) and traffic (bottom) for CORD, source ordering (SO), message passing (MP) and source ordering with write-back policy (WB) for end-to-end workloads (§5.2). CORD outperforms SO by 28% and 20% and is 4% and 2% close to MP performance for CXL and UPI, respectively. CORD reduces SO traffic by 11% and 16% and is 7% and 5% close to MP traffic for CXL and UPI, respectively. Y-axes are normalized to CORD.

evaluate CORD's benefits for interconnect technologies with shorter latency, we also model Intel UPI [33] latency with the same system setup.

Workloads. We evaluate multi-PU workloads from multiple benchmark suites with diverse usage of write-through stores/atomics over release consistency, as summarized in Table 2. We use the Pannotia [16] and Chai [30] benchmarks since they are widely used in prior works on multi-PU cache coherence protocols [8, 9, 57]. We also use the U.S. Department of Energy (DOE) mini-apps — scientific computing workloads that use MPI primitives [66] — to evaluate CORD against message passing. We port MPI primitives to release-consistent shared memory using Relaxed and Release write-through stores. We evaluate DOE mini-apps using traces since their source code and binaries are unavailable.

Compared protocols. We compare CORD against three design schemes for multi-PU release consistency: (*i*) source-ordered write-through cache coherence protocols (SO); (*ii*) message passing (MP); and, (*iii*) source-ordered write-back cache coherence protocols (WB)[2]. For all compared cache coherence protocols, we use their corresponding MESI-based implementation provided by Spandex [9], a state-of-the-art multi-PU cache coherence protocol supporting flexible cache request interfaces. Our CORD implementation is also MESI-based. For MP, we simulate the PCIe protocol's read and write transactions with point-to-point release consistency ordering. For CORD and MP, the look-up table storage is provisioned as summarized in Table 3.

[2]We refer to each scheme by its shorthand for the rest of this section.

5.2 End-to-end Workloads

Performance. Fig. 7 (top) shows that CORD consistently outperforms SO across most workloads over both CXL and UPI. With CXL, CORD outperforms SO by over 28% on average, while with UPI, even though its benefits decrease, CORD still outperforms SO by over 20%. CORD observes significant performance benefits for the DOE workloads (i.e., 20–64% faster than SO for MOCFE, CMC-2D, BigFFT, and CR) because their communication-to-computation ratios are higher than other workloads.

Compared to MP, CORD maintains release consistency across all CPU hosts at less than 4% and 2% average performance overheads with CXL and UPI, respectively. Out of the seven workloads (`TQH` cannot run with MP as we explained in §3.2), CORD performs more than 1% worse than MP only for `TRNS`, `MOCFE`, and `CMC-2D`. This is because their high communication fanout frequently triggers CORD's inter-directory notification mechanism to ensure cross-PU release consistency and because their relatively fine-grained synchronization cannot subsume the latency incurred by such notifications.

WB observes lower performance than CORD for all workloads except for `PR`. `PR` simultaneously exhibits moderate locality so that WB can benefit performance with data reuse and employs relatively coarse-grained synchronization, subsuming the high latency of ordering WB stores at the source (same as SO).

Traffic. Fig. 7 (bottom) shows that CORD reduces the inter-PU traffic compared to SO across various workloads over both CXL and UPI. In particular, CORD reduces traffic compared to SO by over 11% and 16% for CXL and UPI, respectively. In particular, CORD reduces traffic significantly for `PR`, `PAD`, `CMC-2D`, `BigFFT` and `CR` workloads (59%

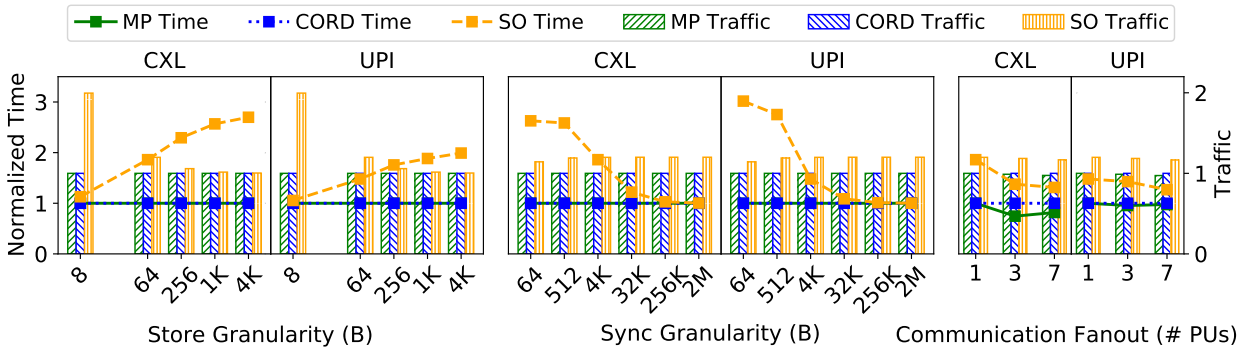

Figure 8: Sensitivity to store granularity, synchronization granularity, and communication fanout (§5.3). Execution time (left y-axis) and traffic (right y-axis) values are normalized by CORD's while x-axes are in log scale. When analyzing sensitivity to a parameter, we fix the other parameter values: store granularity to 64B, synchronization granularity to 4KB, and fanout to 1.

less traffic for PR, 11% – 29% for others). This is because they either employ fine Relaxed store granularity, making acknowledgment messages a significant traffic overhead (*e.g.*, PR and BigFFT), or employ infrequent, coarse-grained synchronization (i.e., coarse Release granularity in Table 2), making CORD's traffic overheads caused by the inter-directory notification messages small relative to the total traffic. In contrast, TRNS and MOCFE are the only workloads for which CORD generates more traffic than SO. This is because their fine-grained synchronization and high communication fanout trigger a high volume of inter-directory notifications, offsetting the traffic savings from eliminating write-through acknowledgments.

Compared to MP, CORD incurs 7% and 5% additional traffic on average over CXL and UPI, respectively. Specifically, out of the seven workloads for which MP can provide release consistency, CORD generates > 5% additional traffic only for two workloads (PAD and MOCFE) due to their fine-grained synchronization and medium to high communication fanout, as explained above.

WB generates significantly more traffic than CORD for all workloads except for SSSP because SSSP with its input graph wing is the only workload that simultaneously exhibits moderate locality so that WB can reduce traffic with data reuse, and employs relatively coarse-grained synchronization so that CORD's traffic saving from acknowledgments becomes insignificant.

5.3 Sensitivity Analysis

We conduct sensitivity analysis along three application characteristics: *(i)* Relaxed store granularity (*e.g.*, word, cache line, or larger), *(ii)* synchronization granularity (i.e., communicated data size per Release store), and *(iii)* communication fanout (i.e., the number of other PUs that each PU communicates with). We consider two characteristics of the system: *(i)* the interconnect latency and *(ii)* CORD's epoch number and store counter bit-width. We use a micro-benchmark that launches a single thread to repeatedly issue write-through stores to other CPU hosts' memory with configurable store granularity, synchronization granularity, and communication fanout. Note that while most multi-PU cache coherence protocols allow at most 64B stores, we measure store granularity to up to 4KB since protocols are visioning larger angularities (*e.g.*, CXL 3.0's 256B flit). For each configuration, we measure the execution time and inter-PU traffic. Unless otherwise specified, store granularity defaults to 64B, synchronization granularity to 2MB, and communication fanout to 1 CPU host.

Relaxed **store granularity.** Fig. 8 (left) shows that with larger Relaxed store granularity (up to 4KB), CORD's performance improvements over SO keep increasing — achieving 63% and 50% lower execution time over CXL and UPI, respectively. This is because larger Relaxed stores increase data transmission efficiency since they amortize packetization overheads seen in smaller stores. In contrast, CORD reduces the execution time by one interconnect round-trip regardless of Relaxed store granularity. As such, CORD's relative performance benefit increases with larger Relaxed store granularity. At the same time, CORD's traffic reduction decreases with larger Relaxed store granularities, reducing to < 10% for 1*KB* stores. This is because SO's acknowledgment messages become a smaller fraction of total traffic at larger Relaxed store granularities. Moreover, CORD observes no performance or traffic overheads compared to message passing since the inter-directory notification is never triggered at communication fanout of 1.

Synchronization granularity. Fig. 8 (middle) shows that at larger synchronization granularities (up to 2MB), CORD's performance benefits over SO decrease, dropping to < 20% over CXL and UPI at 256KB. This is because data transmission time dominates the overall execution time at larger synchronization granularities, minimizing the impact of acknowledgment messages. CORD's traffic reduction over SO first increases slightly with fine-grained synchronization (i.e., 64B), then remains constant at 20% with larger synchronization granularity. This is because, with fine synchronization granularities, both Relaxed and Release stores contribute substantially to the total traffic. CORD, however, cannot eliminate the Release store acknowledgments, resulting in lower amortized traffic reduction. At larger granularity where traffic is dominated by Relaxed stores, CORD reliably reduces traffic proportional to the communicated data size. As before, CORD observes no performance and traffic overheads compared to MP since there are no notification messages.

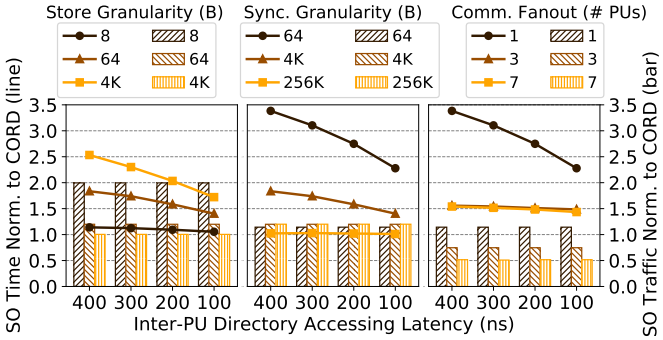

Figure 9: SO's execution time normalized to CORD (line plot, left y-axis) and traffic normalized to CORD (bar plot, right y-axis) with different inter-PU directory access latency, under various application parameters (§5.3).

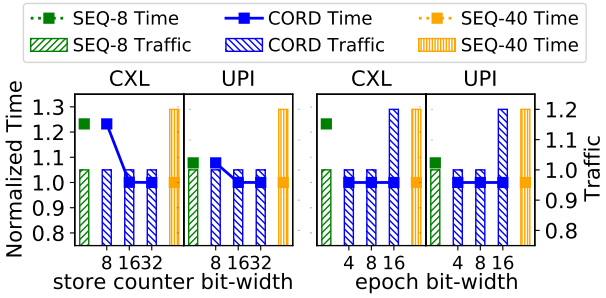

Figure 10: CORD's epoch and store counter vs. sequence number (§5.3). x-axes are log-scale. Execution time (left y-axis) and traffic (right y-axis) are normalized to SEQ-40 and SEQ-8. The epoch number is fixed at 8 bits in (left), and the store counter is fixed at 32 bits in (right).

Communication fanout. Fig. 8 (right) shows that with larger communication fanout (up to 7 CPU hosts), CORD observes a decreasing but significant performance benefit (> 25% and > 20% over CXL and UPI at 7 hosts, respectively) compared to SO. This is because, at higher communication fanouts, CORD's additional inter-directory notification messages delay the Release stores, reducing the overall performance benefit. However, while more notification messages are generated, these messages do not incur much traffic overhead with synchronization granularity of 4KB. As a result, the traffic overhead remains constant at 20%.

Over CXL, CORD observes 20% performance overheads compared to MP at 7 hosts for the same reason its performance benefits over SO decrease. However, it observes < 5% performance overheads compared to MP over UPI, the shorter latency significantly lowering performance degradation caused by inter-directory notifications. Finally, CORD observes ~ 5% traffic overhead compared to MP for the same reason its traffic reduction compared to SO remains constant.

Impact of inter-PU directory access latency. Fig. 9 shows that while with shorter inter-PU latency CORD's execution time improvement over SO diminishes, CORD still observes reduced execution

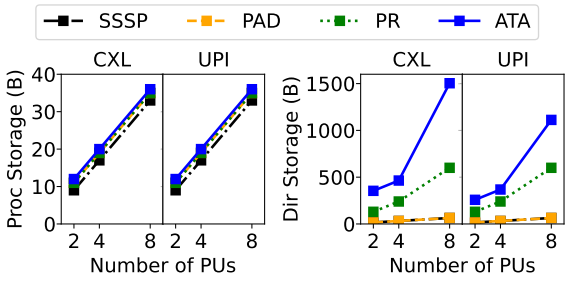

Figure 11: CORD storage overhead for real-world and synthetic storage-hungry workloads (§5.4). CORD incurs negligible storage overhead at the processor and directory, even scaling sub-linearly to CPU hosts.

time across various workload characteristics, because compared to SO, CORD can always reduce interconnect hops in the critical path regardless of workload characteristics. In contrast, CORD's traffic relative to SO remains constant with shorter inter-PU latency since both protocols' interconnect messages to send are unaffected by latency. We find that CORD generates more inter-PU traffic than SO with synchronization-heavy, high-communication-fanout workloads such as the 7-PU configuration in Fig. 9 (right).

Impact of epoch and store counter bit-width. By employing sufficiently large bit-width store counter (i.e., ≥ 16bits) and small bit-width epoch (i.e., ≤ 8bits), CORD can balance both performance and traffic overheads. In Fig.10, SEQ-8 and SEQ-40 represent the baseline 8-bit and 40-bit sequence number ordering approach discussed in §4.1. CORD can simultaneously match the performance of large bit-width sequence number (i.e., `SEQ-40`) as processor stalls for overflow handling are rare and match the traffic of small bit-width sequence number (i.e., `SEQ-8`) as small epoch numbers do not significantly inflate traffic overheads.

5.4 Overheads of CORD

Storage. We focus on three workloads that require the most storage (`SSSP`, `PAD`, and `PR`) along with a synthetic workload that continuously issues the MPI `alltoall` primitive to broadcast 8B data (dubbed `ATA`). `ATA` represents an extreme workload that consumes extremely high storage due to its high communication fanout and very fine-grained synchronization.

As discussed in §4.3, CORD adds hardware storage to track protocol states at the processor and directory using look-up tables. While CORD can operate correctly with any (≥ 1) number of entries in each table, under-provisioning table sizes can result in performance degradation. Our storage overhead results (Fig. 11 and Table 3) show the smallest amount of storage required to ensure no performance degradation; this is the amount of storage provisioned across all other evaluations.

Fig. 11 shows that the processor storage not only increases sub-linearly with CPU hosts but is also negligible (< 40B) for all workloads over CXL and UPI. While the directory storage overhead increases linearly with more CPU hosts, the most storage-consuming workload, `ATA`, only consumes < 1.5KB and ~ 1.2KB at 8 hosts for

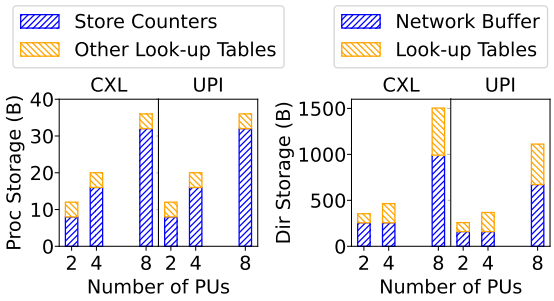

Figure 12: Storage overhead breakdown for ATA (§5.4). Store counters dominate storage at the processor, while network buffers and look-up tables both contribute significantly to storage and scale sub-linearly at the directory.

Component	Size (entries)	Area (mm^2)	Power (mW)	Acc. Energy (r/w nJ)
Processor (total)		0.066	9.242	
store counter	8	0.033	4.621	0.016/0.016
unAck-ed epoch	8	0.033	4.621	0.016/0.016
Directory (total)		0.136	23.454	
store counter	8*16	0.045	7.776	0.017/0.021
notification counter	16*16	0.058	11.057	0.017/0.025
largest Comm. epoch	8	0.033	4.621	0.016/0.017

Table 3: Look-up table sizes; area and power overheads.

CXL and UPI, respectively. In comparison, each CPU host employs 8 slices of 2MB LLC, which is four orders of magnitude larger.

Fig. 12 breaks down the storage for ATA, the synthetic workload consuming extremely high storage. At the processor, the store counters consume most of the storage and scale sub-linearly with more hosts, as it is maintained per directory. The remaining table for tracking unacknowledged epochs only contributes to a small constant portion of storage, as its required entries for holding unacknowledged Release stores and depend only on the application's synchronization granularity and interconnect latency, independent of the number of hosts. At the directory, both look-up tables and network buffers to hold recycled Release stores contribute significantly to total storage. Both overheads grow sub-linearly with more CPU hosts. This is because both the look-up tables and network buffers' storage depend directly on the total number of recycled Release stores at the current directory, which scales sub-linearly with more CPU hosts issuing the MPI `alltoall` primitive.

Area and power. We used CACTI 7.0 [11] to estimate CORD's area and power overheads at the processor core and directory, assuming the 22nm technology. As shown in Table 3, CORD incurs 0.066mm^2 area and 9.242mW static power at each processor core, two orders of magnitude smaller than that by a typical server CPU core [15, 44, 65]. At the directory, CORD incurs 0.136mm^2 area and 23.454mW static power. In comparison, each CPU host's LLC slices and cache directories consume 82.642mm^2 area and 1761.256mW static power as estimated by CACTI 7.0, i.e., CORD's power and area overheads at the directory are less than 0.2% and 1.3%, respectively.

Our estimated dynamic energy needed to access the lookup tables ranges from 0.016–0.025nJ. In contrast, CXL 3.0 and PCIe 6.0's energy use is estimated at 4–5pJ/bit [19, 61] (i.e., 2–2.5nJ for 64B) while writing a 64B cache line into the LLC uses 3.407nJ as estimated by CACTI 7.0, i.e., CORD's dynamic energy overhead to transmit a 64B store is < 1% of total energy consumed.

6 CORD for TSO memory model

While CORD targets release consistency, we study its impact on workload performance and interconnect traffic under the Total Store Ordering (TSO) memory model used in x86 systems.

We adopt the same simulation configuration (Table 1) and workloads (Table 2) as previous evaluations, modifying all coherence protocols to enforce TSO rather than release consistency. TSO [51] preserves *program-order* across all memory accesses except between a preceding store and a subsequent load due to the presence of FIFO store buffers. In our implementation, we modify SO and WB to source-order all memory accesses rather than only for Acquire and Release in release consistency, and add a FIFO store buffer in our simulated processor. We modify CORD to enforce store-store ordering for all write-through stores with the Release-Release ordering mechanism as described in §4.1, while other memory accesses are still source-ordered (see §4.4). While message-passing protocols are not known to enforce TSO, we modify MP to totally order all simulated PCIe read and write transactions as an upper bound for performance and traffic efficiency. Under TSO:

CORD retains its performance improvements but observes more traffic compared to SO. Fig. 13 (top) shows that under TSO, CORD outperforms SO across most workloads: with CXL, CORD outperforms SO by over 102% on average, and with UPI, CORD outperforms SO by over 73%. TSO must order all stores, providing CORD with significant improvement opportunities by efficiently enforcing write-through ordering. However, Fig. 13 (bottom) shows that CORD observes higher traffic than SO except for SSSP and PAD, in contrast to *reducing* traffic for most workloads under release consistency. This is because CORD must add (1) acknowledgments for write-through stores under TSO as they must be totally ordered using the same mechanism as Release stores under release consistency (§4.1), and (2) inter-directory notification messages (§4.2).

CORD, SO and WB observe higher traffic overhead over MP. Most notably, for PR, MP incurs only 24% and 32% of the traffic incurred by CORD with CXL and UPI, respectively. This is because MP's more relaxed point-to-point ordering model allows it to eliminate acknowledgments for totally ordered stores, while all other compared cache coherence protocols require them to enforce TSO globally.

7 Related Works

We discussed prior approaches for release consistency in multi-PU systems in §3; we now focus on other research related to CORD.

Cache-coherent Multi-PU interconnects. Multi-PU systems in industry [20, 21, 24, 63] and academia [5, 38, 57, 60] employ cache-coherent interconnects for application performance and system efficiency with flexible and simple programming models. Industry

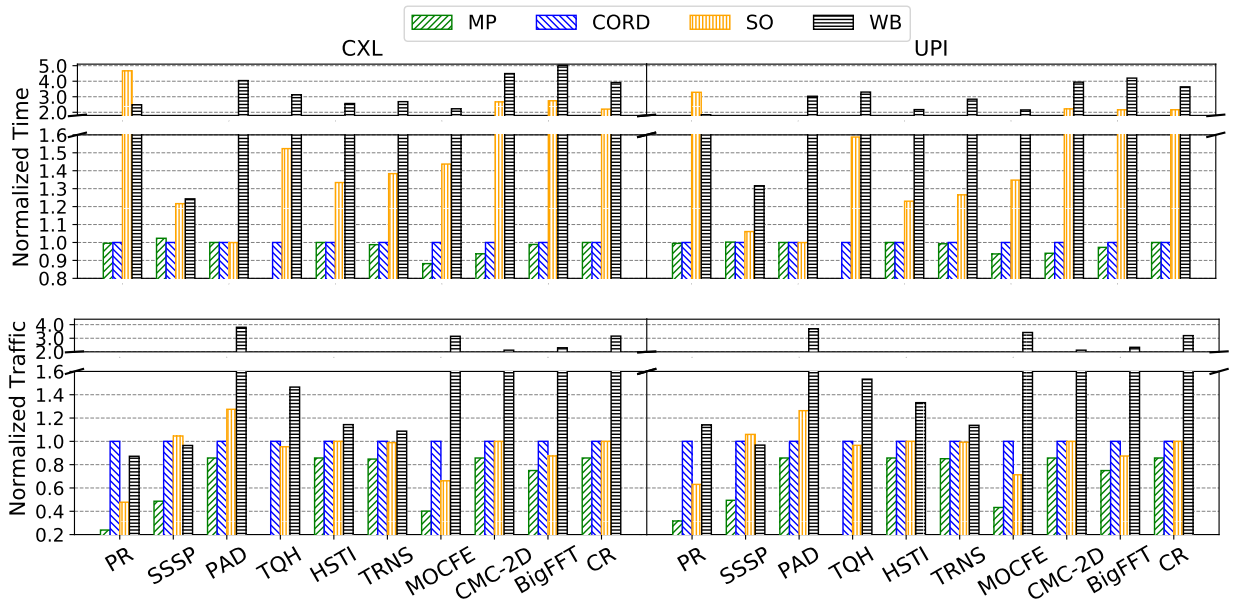

Figure 13: Performance (top) and traffic (bottom) under TSO for end-to-end workloads (§6). CORD outperforms SO by 102% and 73% and is 3% and 3% close to MP performance for CXL and UPI, respectively, but inflates SO traffic by 8% and 6% for CXL and UPI, respectively. Y-axes are normalized to CORD.

standards include NVIDIA NVLink-C2C [23], AMD Infinity Fabric [63], Arm AMBA CHI [1], CXL [24], Gen-Z (merged to CXL), CCIX [3], and IBM CAPI [64]. Recent academic works have also explored cache-coherence across CPUs using programmable networks [38]. Our work builds atop these interconnect technologies.

Heterogeneous cache coherence protocols. Prior works have extensively explored heterogeneous cache coherence design across multiple PUs, focusing on flexibility [9], performance [31, 37, 56, 68], hardware overhead [18, 36] and design/verification complexity [13, 18, 47, 50]. However, they either omit write-through policy [18, 50], or support write-through but resort to source ordering when enforcing certain memory models [9, 13, 31, 36, 37, 47, 56, 68]. Our work aims to improve these proposals for efficient release consistency with write-through coherence.

Heterogeneous memory models across multiple PUs can be classified based on whether or not they introduce the notion of *scope*, where a scope defines a cohort of threads within which memory consistency is enforced. The HSA foundation's heterogeneous data-race-free (HRF) [6, 32] and NVIDIA's PTX [42] memory model introduce scopes, while the compound memory model [28, 46, 50] does not. Although our work focuses on release consistency for its popularity in these models, it is generalizable for efficiently enforcing other consistency models for write-through coherence.

8 Conclusion

Emerging multi-PU unit systems employ release-consistent shared memory with write-through coherence. We have shown that the de facto approach of ordering write-through accesses at the source processor results in unnecessary application slowdowns and interconnect traffic overheads, while message passing can result in release consistency violations without careful orchestration. We presented CORD, a novel cache coherence protocol that orders write-through accesses at the directory for release consistency to improve performance and reduce traffic. Compared to source ordering, CORD improves performance by 23% and reduces traffic by 16% at < 1% storage, area, and power overheads.

Acknowledgments

We thank the anonymous ISCA reviewers for their valuable comments and insightful feedback. We would also like to thank David Nellans, Guillermo Juan Rozas, Gonzalo Brito Gadeschi, and Daniel Lustig for their feedback on various aspects of this work. This work was supported in part by the NSF's awards 2047220, 2112562, 2147946, and a NetApp Faculty Fellowship.

References

[1] 2024. Arm AMBA CHI specification. https://developer.arm.com/documentation/ihi0050/latest/.
[2] 2024. PCIe specifications. https://pcisig.com/specifications.
[3] 2025. CCIX Consortium. https://www.ccixconsortium.com/.
[4] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. 1988. An Evaluation of Directory Schemes for Cache Coherence. In *Proc. ACM/IEEE ISCA*.
[5] Neha Agarwal, David Nellans, Eiman Ebrahimi, Thomas F Wenisch, John Danskin, and Stephen W Keckler. [n. d.]. Selective GPU caches to eliminate CPU-GPU HW cache coherence. In *Proc. IEEE HPCA*.
[6] Jade Alglave and Luc Maranget. 2016. *Towards a Formalization of the HSA Memory Model in the cat Language.* Technical Report. HSA Foundation. https://hsafoundation.com/wp-content/uploads/2021/02/cat_ModelExpressions-1.1.pdf
[7] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* (2014).

[8] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. 2016. Lazy release consistency for GPUs. In *Proc. IEEE/ACM MICRO*.

[9] Johnathan Alsop, Matthew Sinclair, and Sarita Adve. 2018. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *Proc. ACM/IEEE ISCA*.

[10] James Archibald and Jean-Loup Baer. 1986. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Trans. Comput. Syst.* (1986).

[11] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* (2017).

[12] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. *SIGPLAN Not.* (2011).

[13] Jesse G. Beu, Michael C. Rosier, and Thomas M. Conte. 2011. Manager-client pairing: A framework for implementing coherence hierarchies. In *Proc. IEEE/ACM MICRO*.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* (2011).

[15] John Byrne, Jichuan Chang, Kevin T. Lim, Laura Ramirez, and Parthasarathy Ranganathan. 2011. Power-efficient networking for balanced system designs: early experiences with PCIe. In *Proc. 4th Workshop on Power-Aware Computing and Systems*.

[16] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *Proc. IEEE International Symposium on Workload Characterization (IISWC)*.

[17] Linchuan Chen, Xin Huo, and Gagan Agrawal. 2012. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proc. IEEE/ACM SC Conf.*

[18] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proc. 2011 International Conference on Parallel Architectures and Compilation Techniques*.

[19] Dong-Myung Choi, Yikui Dong, Roan Nicholson, Frank Liu, Wenyan Jia, Vadim Levin, Mike He, Sameer Pradhan, Jieqiong Du, Michael De Vita, Amanda Tran, Reza Navid, Sitaraman Iyer, and Rui Song. 2024. A 4.6pJ/b 64Gb/s Transceiver Enabling PCIe 6.0 and CXL 3.0 in Intel 3 CMOS Technology. In *Proc. 2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*.

[20] NVIDIA Corporation. 2024. NVIDIA Grace Blackwell Superchip. https://nvidianews.nvidia.com/news/nvidia-blackwell-platform-arrives-to-power-a-new-era-of-computing.

[21] NVIDIA Corporation. 2024. NVIDIA Grace Hopper Superchip. https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper.

[22] NVIDIA Corporation. 2024. NVIDIA NVLink. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/.

[23] NVIDIA Corporation. 2024. NVIDIA NVLink-C2C. https://www.nvidia.com/en-us/data-center/nvlink-c2c/.

[24] CXL Consortium. 2022. CXL 3.0 Specification. https://www.computeexpresslink.org/download-the-specification.

[25] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* (2024).

[26] David L Dill. 1996. The Mur ϕ verification system. In *Proc. Computer Aided Verification: 8th International Conference*.

[27] J. Dorsey, Shawn Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. 2007. An Integrated Quad-Core Opteron Processor. In *IEEE International Solid-State Circuits Conference*.

[28] Andrés Goens, Soham Chakraborty, Susmit Sarkar, Sukarn Agarwal, Nicolai Oswald, and Vijay Nagarajan. 2023. Compound Memory Models. *Proc. ACM Program. Lang.* (2023).

[29] J. Goodman and H. Hum. 2004. *MESIF: A two-hop cache coherency protocol for point-to- point interconnects.* Technical Report 2004-002. Department of Computer Science, University of Auckland.

[30] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Víctor García-Floreszx, Simon Garcia de Gonzalo, Thomas B. Jablin, Antonio J. Peña, and Wen-mei Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[31] Blake A. Hechtman, Shuai Che, Derek R. Hower, Yingying Tian, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In *Proc. IEEE HPCA*.

[32] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2014. Heterogeneous-race-free memory models. In *Proc. ACM ASPLOS*.

[33] Intel Corporation. 2024. An Introduction to the Intel QuickPath Interconnect. https://www.intel.ca/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

[34] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: efficient CPU-GPU co-execution for deep learning inference on mobile devices.. In *Proc. ACM MobiSys*.

[35] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters. In *Proc. USENIX OSDI*.

[36] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. 2016. Building Heterogeneous Unified Virtual Memories (UVMs) without the Overhead. *ACM Trans. Archit. Code Optim.* (2016).

[37] Snehasish Kumar, Arrvindh Shriraman, and Naveen Vedula. 2015. Fusion: Design tradeoffs in coherent cache hierarchies for accelerators. In *Proc. ACM/IEEE ISCA*.

[38] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. Mind: In-network memory management for disaggregated data centers. In *Proc. ACM SOSP*.

[39] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proc. ACM ASPLOS*.

[40] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. 2015. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *Proc. 2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*.

[41] Bin Liu, Dawid Zydek, Henry Selvaraj, and Laxmi Gewali. 2012. Accelerating high performance computing applications: Using cpus, gpus, hybrid cpu/gpu, and fpgas. In *Proc. 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*.

[42] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *Proc. ACM ASPLOS*.

[43] Michal Marks and Ewa Niewiadomska-Szynkiewicz. 2014. Hybrid CPU/GPU Platform For High Performance Computing.. In *ECMS*.

[44] Michael McKeown, Alexey Lavrov, Mohammad Shahrad, Paul J. Jackson, Yaosheng Fu, Jonathan Balkind, Tri M. Nguyen, Katie Lim, Yanqi Zhou, and David Wentzlaff. 2018. Power and Energy Characterization of an Open Source 25-Core Manycore Processor. In *Proc. IEEE HPCA*.

[45] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Comput. Surv.* (2015).

[46] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, David A. Wood, and Natalie Enright Jerger. 2020. *A Primer on Memory Consistency and Cache Coherence*.

[47] Lena E. Olson, Mark D. Hill, and David A. Wood. 2017. Crossing Guard: Mediating Host-Accelerator Coherence Interactions. In *Proc. ACM ASPLOS*.

[48] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. 2018. ProtoGen: Automatically generating directory cache coherence protocols from atomic specifications. In *Proc. ACM/IEEE ISCA*.

[49] Nicolai Oswald, Vijay Nagarajan, and Daniel J Sorin. 2020. HieraGen: Automated generation of concurrent, hierarchical cache coherence protocols. In *Proc. ACM/IEEE ISCA*.

[50] Nicolai Oswald, Vijay Nagarajan, Daniel J Sorin, Vasilis Gavrielatos, Theo Olausson, and Reece Carr. 2022. HeteroGen: Automatic synthesis of heterogeneous cache coherence protocols. In *Proc. IEEE HPCA*.

[51] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better x86 memory model: x86-TSO. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*.

[52] Saptadeep Pal, Eiman Ebrahimi, Arslan Zulfiqar, Yaosheng Fu, Victor Zhang, Szymon Migacz, David Nellans, and Puneet Gupta. 2019. Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training. *Proc. IEEE/ACM MICRO* (2019).

[53] Mark S. Papamarcos and Janak H. Patel. 1984. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. ACM/IEEE ISCA*.

[54] Adarsh Patil, Vijay Nagarajan, Nikos Nikoleris, and Nicolai Oswald. 2023. Āpta: Fault-tolerant object-granular CXL disaggregated memory for accelerating FaaS. In *Proc. IEEE/IFIP DSN*.

[55] David A. Patterson and John L. Hennessy. 1990. *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers Inc.

[56] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. 2013. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proc. IEEE/ACM MICRO*.

[57] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. 2020. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *Proc. IEEE HPCA*.

[58] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors *(Proc. ACM PLDI)*.

[59] Gabin Schieffer, Jacob Wahlgren, Jie Ren, Jennifer Faj, and Ivy Peng. 2024. Harnessing Integrated CPUg-GPU System Memory for HPC: a first look into Grace Hopper. In *Proc. ICPP*.

[60] Henry N. Schuh, Arvind Krishnamurthy, David Culler, Henry M. Levy, Luigi Rizzo, Samira Khan, and Brent E. Stephens. 2024. CC-NIC: a Cache-Coherent

Interface to the NIC. In *Proc. ACM SOSP*.
[61] Debendra Das Sharma. 2024. PCI-Express: Evolution of a Ubiquitous Load-Store Interconnect Over Two Decades and the Path Forward for the Next Two Decades. *IEEE Circuits and Systems Magazine* (2024).
[62] Nikolay A Simakov, Matthew D Jones, Thomas R Furlani, Eva Siegmann, and Robert J Harrison. 2024. First Impressions of the NVIDIA Grace CPU Superchip and NVIDIA Grace Hopper Superchip for Scientific Workloads. In *Proc. International Conference on High Performance Computing in Asia-Pacific Region Workshops*.
[63] Alan Smith, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Samuel Naffziger, Mike Mantor, Mark Fowler Nathan Kalyanasundharam, Vamsi Alla, Nicholas Malaya, Joseph L. Greathouse, Eric Chapman, and Raja Swaminathan. 2024. Realizing the AMD Exascale Heterogeneous Processor Vision : Industry Product. In *Proc. ACM/IEEE ISCA*.
[64] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development* (2015).
[65] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David R. Kaeli. 2019. Summarizing CPU and GPU Design Trends with Product Data. *CoRR* (2019).
[66] U.S. Department of Energy. 2014. Characterization of the DOE Mini-apps. https://portal.nersc.gov/project/CAL/doe-miniapps.htm.
[67] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. 2022. Evaluating emerging CXL-enabled memory pooling for HPC systems. In *Proc. 2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*.
[68] Moyang Wang, Tuan Ta, Lin Cheng, and Christopher Batten. 2020. Efficiently Supporting Dynamic Task Parallelism on Heterogeneous Cache-Coherent Systems. In *Proc. ACM/IEEE ISCA*.
[69] Chenyang Zhang, Feng Zhang, Xiaoguang Guo, Bingsheng He, Xiao Zhang, and Xiaoyong Du. 2020. iMLBench: A machine learning benchmark suite for CPU-GPU integrated architectures. *IEEE Transactions on Parallel and Distributed Systems* (2020).
[70] 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 *Proc. ACM SOSP*.
[71] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale heterogeneous graphs. In *Proc. 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*.

A Artifact Appendix

A.1 Abstract

There are three artifacts for this paper: (1) a Gem5 implementation with corresponding benchmarks, datasets, and scripts to reproduce Figures 2, 7, 8, 9, 10, 11, 12, and 13, (2) a Cacti script to reproduce Table 3, and (3) a Murphi litmus test suite for model-checking protocol correctness (§4.5). The artifact requires a Linux server with 150 GB of disk space, with Docker and Python 3.6+ installed.

A.2 Artifact check-list (meta-information)

- **Program:** Gem5, Murphi, Cacti, Docker
- **Run-time environment:** Ubuntu 18.04.5 or higher, Docker, Python3.6 or higher, gcc 7.5.0.
- **Hardware:** A Linux System with Intel X86-64 CPU
- **Metrics:** Execution time, traffic, storage, area, power, and energy
- **Output:** Figures and tables in the paper, and the corresponding raw experimental data
- **Experiments:** A Docker image (for Gem5 experiments) and scripts are provided to reproduce results. Gem5 experiments may have minimal (1%-2%) differences due to randomization.
- **How much disk space required (approximately)?:** 150 GB
- **How much time is needed to prepare workflow (approximately)?:** 1-1.5 hours
- **How much time is needed to complete experiments (approximately)?:** 1-1.5 hours
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to access. `https://github.com/yale-nova/CORD`

A.3.2 Hardware dependencies. A Linux System with Intel X86-64 CPU and about 150 GB of disk space. Our experiments were conducted on a server with 96 cores and 376 GB of RAM. Servers with fewer cores or RAM should work but may result in longer runtimes than reported in this paper.

A.3.3 Software dependencies. Ubuntu 18.04.5 or higher, Docker, Python 3.6 or higher (with matplotlib, numpy, pandas), gcc 7.5.0 (higher may work).

A.4 Installation

First, clone the artifact repository:

```
$: git clone https://github.com/yale-nova/CORD
```

Then, in the main directory of the repository, execute:

```
$: ./setup.sh
```

This script will download a docker image (~ 15 mins), launch a Docker container for Gem5 experiments (~ 5 mins), and compile the Gem5, Cacti, and Murphi implementations (~ 50 mins). This script is expected to take 1-1.5 hours. You should see `Setup Complete` upon completion.

A.5 Experiment workflow

In the main directory of the repository, execute:

```
$: ./run.sh
```

This script will run all Gem5 (~ 1 hour) and Cacti (~ 1 sec) evaluations, then run all Murphi litmus tests (~ 5 mins). This script is expected to take 1-1.5 hours. You should see a:

```
All experiments complete
```

message upon completion.

A.6 Evaluation and expected results

In the main directory of the repository, execute:

```
$: ./gen_figures.sh
```

This script will first process all Gem5 and Cacti evaluation results and place the processed result under `plots/` in `csv` format (~ 1 min), then plot the processed results and generate the original `pdf` figures and `csv` tables under `figures/` (~ 1 min). This script is expected to take 2 minutes. You should see `Artifact evaluation complete` upon completion.

The generated Gem5 experiment results (Figures 2, 7, 8, 9, 10, 11, 12, and 13) may have small (1%-2%) differences with the original figure in the paper due to simulator randomization. The generated Cacti experiment result (Table 3) should be identical to that in the paper. The Murphi model checking is passed if `run.sh` outputs `Murphi for model checking complete`.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae