Understanding and Profiling CXL.mem Using PathFinder

Xiao Li[1,3], Zerui Guo[1], Yuebin Bai[3], Mahesh Ketkar[2], Hugh Wilkinson[2], Ming Liu[1]

1.*University of Wisconsin-Madison* 2.*Intel* 3.*Beihang University*

Abstract

CXL.mem and the resulting memory pool are promising and gaining great attention. Unlike local memory, CXL DIMMs stay at the I/O subsystem, whose inferior performance can easily impact the processor pipeline and memory subsystem, yielding performance interference, hardware contention, obscure behaviors, and underutilized communication and computing resources. However, our community lacks a tool to understand and profile the CXL.mem protocol execution end-to-end between CPU and remote DIMM.

This paper fills the gap by designing and implementing **PathFinder**[1], a systematic, informative, and lightweight CXL.mem profiler. PathFinder leverages the capabilities of existing hardware performance monitors (PMUs) and dissects the CXL.mem protocol at adequate granularities. Our key idea is to view the server processor and its chipset as a multi-stage Clos network, equip each architectural module with a PMU-based telemetry engine, track different CXL.mem paths, and apply conventional traffic analysis techniques. PathFinder performs snapshot-based path-driven profiling and introduces four techniques, i.e., path construction, stall cycle breakdown, interference analyzer, and cross-snapshot analysis. We build PathFinder atop Linux Perf and apply it to seven case studies.

CCS Concepts

• Networks → Network performance analysis; **• Hardware → Buses and high-speed links**; **• Software and its engineering** → *Abstraction, modeling and modularity*;

Keywords

CXL, Memory Pooling, Performance Profiling

ACM Reference Format:
Xiao Li[1,3], Zerui Guo[1], Yuebin Bai[3], Mahesh Ketkar[2], Hugh Wilkinson[2], Ming Liu[1]. 2025. Understanding and Profiling CXL.mem Using PathFinder. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3718958.3750479

1 Introduction

CXL-based memory disaggregation [6, 10, 15, 16, 19–21, 24, 25] has gained significant traction recently due to independent resource scaling, high utilization, and cost efficiency. We have seen a growing interest in deploying CXL memory pooling in data centers, enterprise clusters, and edge clouds. With the next-generation CXL fabric and the capability to build multi-tiering fabric switching [18], a disaggregated memory pool can provide tens to hundreds of terabytes of capacity at the bandwidth close to local DIMMs.

The key technology enabler is the CXL.mem protocol [18] that allows host processors to issue load/store instructions directly to remote DIMMs. A memory request traverses the processor pipeline, served by its local memory hierarchy first (L1D, L2, and LLC) and then forwarded to the CXL DIMM via the FlexBus. The data response is then delivered to the memory subsystem and resumes the stalled pipeline execution. Load/store commands traverse over the underlying fabric as flits, completely transported to the host. The remote memory usually exposes itself as a CPUless NUMA node, facilitating building memory tiering and object-based remote memory solutions [46, 66, 79, 100, 108]. This significantly simplifies porting existing applications and developing new ones.

However, unlike local memory, accessing CXL memory is inherently slow because requests must traverse the system bus or even cluster interconnect in a switched pooling case. This not only slows down the application execution, but more importantly, stalls the processor pipeline and changes the memory subsystem's access characteristics. For example, some micro-architecture components (like line fill buffer) would be congested, blocking local memory requests. The data locality and application working set become unpredictable when switching from local to CXL memory because read/prefetch-induced CXL loads take longer to fetch data, whereas more local requests are issued, competing for the available slots in different caching layers. Besides, one would experience underutilized communication and computing resources because there might exist some contention along the CPU pipeline, stymying from submission of more memory commands. Until now, *our community lacks a systemic profiling tool to understand and analyze the* CXL.mem *protocol execution and unearth the root causes of the above scenarios.*

Building such a utility is non-trivial and entails three challenges. First, CXL.mem has multiple non-transparent data paths (§2.2) between cores and off-chip memory, whose execution characteristics hinge on compute, memory, and I/O substrate. Second, CXL.mem load/store instructions are tightly coupled with the deep and out-of-order processor pipeline, which runs at the nanosecond-scale granularity and exposes fewer interfaces to query its execution status. Third, the overlapping and intertwining nature between local and CXL memory streams makes it hard to isolate individual behaviors. Today, people mostly tackle this issue in an ad-hoc manner, i.e., combining a list of profiling tools from micro-architecture [1, 3], memory subsystem [13, 43, 80], and PCIe levels [55, 69, 86], with no end-to-end diagnostic capability.

We find that the performance monitor unit (PMU) on commodity servers is powerful and provides a variety of counters to track

[1]PathFinder is available at https://github.com/netlab-wisconsin/PathFinder.

SIGCOMM '25, September 8–11, 2025, Coimbra, Portugal
© 2025 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 979-8-4007-1524-2/2025/09…$15.00
https://doi.org/10.1145/3718958.3750479

different CXL.mem data paths. We explore a slew of PMUs across the core, uncore, interconnect, system bus, and CXL DIMM modules, empirically explore their capabilities and limitations, and identify 232 counters to dissect the CXL.mem protocol execution (§3).

We then design and implement the **PathFinder** utility to systematically characterize and analyze CXL.mem. Our key idea is to view the server processor and its chipset as a multi-stage Clos network, equip each architectural module with a PMU-based telemetry engine, track different CXL.mem paths, and apply conventional traffic analysis techniques [23, 32, 36, 44, 60, 103]. PathFinder performs snapshot-based path-driven profiling–it takes time series snapshots, classifies CXL.mem transactions based on paths, and examines how concurrent paths interleave over each on-path hardware module. PathFinder comprises four techniques: (a) *PFBuilder*, constructing the CXL data path map via synthesizing different PMU counters (§4.3); (b) *PFEstimator*, which employs a back-propagation algorithm that gradually attributes the CXL-induced stall cycles in a bottom-up fashion to the CPU pipeline (§4.4); (c) *PFAnalyzer*, zooming each architectural component via white-box modeling and exploring how concurrent CXL and non-CXL streams interfere with each other (§4.5); (d) *PFMaterializer*, which introduces an internal time-series database, takes a per-snapshot digest as inputs, and identifies consistent execution characteristics, such as data locality, contention, and resource under-utilization (§4.6). In sum, PathFinder dissects CXL.mem accesses and associated applications across the entire server system from the temporal and spatial dimensions at the required granularity.

We build PathFinder atop the Linux Perf and evaluate it on two types of machines: (1) an Intel Sapphire Rapids (SPR) server with an Intel Agilex-based CXL DIMM; (b) an Intel Emerald Rapids (EMR) server with Micron CZ120 CXL DIMMs. Using 77 applications (i.e., SPEC CPU 2017 [17], PARSEC [35], SPLASH-2x [105], GAP [33], and Redis [14]), we demonstrate the effectiveness of PathFinder via seven cases: classifying different CXL.mem-induced paths, breaking down the CPU stall cycles, analyzing the interference between local and CXL memory streams, locating elephant CXL flows, understanding bandwidth allocation among current CXL memory streams, analyzing the data locality, and applying PathFinder for performance optimization. PathFinder is open-source, and we will keep working with the community to improve it.

2 Background and Motivation

2.1 CXL.mem Protocol

CXL [18] is a high-speed interconnect technology based on memory semantics. It is built atop the physical layer of PCI Express (PCIe) [22] and provides the load/store interface for remote memory and accelerator communications. CXL exposes root access points to the host processor via the Flex Bus I/O architecture [18]. It encloses three types of protocols: (a) CXL.cache that enables device data coherence via a processor snoop filter mechanism; (b) CXL.mem that allows direct device memory access from host CPUs via load/store instructions; (c) CXL.io, similar to PCIe with some enhancements, like up to 32 lanes and non-coherent data read/write.

This work focuses on CXL.mem and Type-3 host managed device memory, the basis for memory pooling [34, 45, 46, 65, 66, 72]. In this mode, the coherence engine runs in the host processor, operating as a master, whereas the remote memory device works as a subordinate. There are (a) two CXL.mem request transactions from a master to a subordinate (M2S): Request without data (Req) for read and Request with Data (RwD) for write; (b) two response ones in the reverse direction (S2M): Data Response (DRS) and No Data Response (NDR) for the corresponding read and write return. Upon receiving an M2S request, the device memory controller parses the command, reads/writes to the memory media, and returns an S2M data or completion. Note that CXL.mem (1) supports the 68B, 268B, and PBR (Port-Based Routing) flit modes; (2) allows other coherent models (like device-managed coherence via back-invalidation).

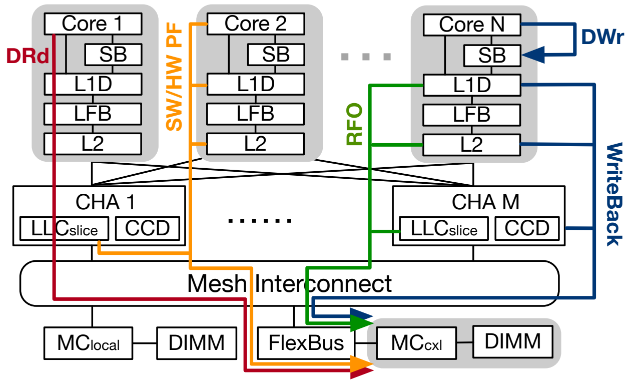


Figure 1: Four CXL.mem data paths on an Intel SPR/EMR processor. SB=Store Buffer. LFB=Line Fill Buffer. CCD=Cache-Coherent Directory. CHA=Caching and Home Agent.

2.2 CXL.mem Data Path

A CXL memory behaves similarly to the local memory and is accessed via load/store instructions from the host processor. In a typical memory subsystem, a memory request is first served from L1, L2, and L3/LLC caches, and then queries the DIMM when missed from the cache hierarchy. Generally, there are four types of architectural requests yielding *CXL.mem* transactions (Figure 1): demand data read (DRd), demand data write (DWr), read for ownership (RFO), and hardware/software prefetching (HW/SW PF). Below, we take the Intel scalable processor as an example for description.

- **#1: DRd → CXL.mem load.** A demand read request first looks up the L1D cache, then submits to the line fill buffer (LFB) when there is a miss. LFB is a per-core hardware FIFO queue with tens of cacheline-sized entries, buffering read responses from its home core. It can serve a DRd when there is a hit and is sent to L2 when missed. The per-core L2 cache works similarly to an L1D and forwards missing DRds to a shared exclusive LLC. An LLC usually comprises a series of equal-sized slices. Each slice is coupled with a cache coherence directory–called Snoop Filter (SF) [9], where both co-locate in a designated Caching and Home Agent (CHA) hardware module. Intel processors run a MESIF-like cache coherent protocol [51]. An LLC hit returns data directly, whereas a miss might trigger either (1) a local snoop to other CHAs within the processor or (2) a cross-socket remote snoop depending on the coherent directory information maintained [12] in the SF. When the DRd cannot be served from caches, it is routed to a local or remote memory controller (MC) via a mesh interconnect [84, 85]. An MC comprises one or several channels that map to an individual DIMM. The CXL memory device connects to the host through the FlexBus I/O subsystem.

- **#2: DWr → RFO → CXL.mem load/store.** A demand write hits the store buffer (SB) first, a specialized FIFO queue with dozens of entries to decouple instruction execution from retirement. Next, it is sent to the L1D cache. Depending on the coherent state, if the data entry is the shared (S), invalid (I), or forward (F) state, a read for ownership (RFO) coherent message is triggered to gain exclusive access (discussed next). The actual CXL.mem store happens when a writeback is issued from L1D, L2, or LLC.
- **#3: RFO → CXL.mem load.** An RFO request follows the same path as a DRd, except originating from the L1D, L2, or LLC. The requesting core would load a data copy with an exclusive (E) state from the other core's caches or main memory.
- **#4: HW/SW PF → DRd/RFO → CXL.mem load.** Data prefetching is a widely used latency-hiding technique. Explicit software prefetching, guided by programs, preloads data from memory, benefiting irregular data structure traversal. Hardware prefetching instead fetches future data implicitly by analyzing and predicting the memory access pattern, performed on a dedicated hardware engine. As shown in Figure 1, Intel processors support L1, L2, and even L3 HW prefetcher (since Sapphire Rapids [4]). Both HW/SW prefetching happens asynchronously and would trigger DRd and RFOs depending on the cache coherence state, causing CXL.mem load transactions for later execution.

2.3 Problem, Challenges, and Prior Solutions

Problem. Unsurprising, the CXL Type-3 memory underperforms local ones. For example, on our SPR server (§5.1), when running the Intel MLC utility [7], the local memory's random access latency and bandwidth are 103.2ns and 131.1GB/s, whereas a NUMA node sustains at 163.6ns and 94.4GB/s. However, a remote CXL DIMM only achieves 355.3ns access latency and 17.6GB/s bandwidth. Such a performance discrepancy not only slows down application execution but, more importantly, stalls the processor pipeline frequently and alters the memory subsystem access characteristics. Specifically, several issues arise (§5). First, it would congest several queue-based architectural components (like SB, LFB, and CHA), where the contention is back-propagated along the CXL.mem data path (§2.2), further affecting other applications accessing local memory. Second, it implicitly changes the application's data locality and working set, especially when multiple CXL and local memory flows co-exist. This is because read/prefetch-induced CXL loads take longer to fetch data into the caching hierarchy. In contrast, more local memory requests are issued concomitantly, competing for available slots at different caching layers. Third, fewer data movements are scheduled due to CPU backpressure and limited memory-level parallelism, even though ample bandwidth is available at the interconnect and system bus, entailing resource underutilization and reduced computing throughput. Therefore, it is pivotal to characterize how CXL memory flows traverse different architectural components end-to-end, localize the system bottlenecks, and generate optimization insights for operators and developers.

Challenges. However, achieving this entails three challenges. First, CXL.mem has several non-transparent data paths (§2.2), originating from core or caches, traversing a chain of on-chip hardware modules non-deterministically (depending on data locality), and finally hitting an off-chip memory device, whose execution characteristics hinge on compute, memory, and I/O substrates. Second, CXL.mem load/store instructions are tightly coupled with the processor execution, which has multiple deep and out-of-order pipelines, runs at the nanosecond-scale granularity, and only exposes a few programmable interfaces to query the execution telemetry information online. Any proactive tracing and instrumental mechanisms are infeasible and prohibitively expensive. Third, the local and CXL memory flows share the underlying hardware, whose data paths are overlapped and intertwined. It is hard to isolate their executions, separate their architectural behaviors and performance impacts, and determine whether CXL memory access is the culprit.

Inadequacies of Prior Solutions. People have tackled this issue by collaboratively applying profiling utilities from different hardware components, carefully synthesizing empirical observations, and approximately inferring the root causes. For example, at the processor level, one would employ micro-architectural profilers (like Intel VTune [1] and AMD uProf [3]) based on a well-established Top-Down Analysis (TMA) technique [104]. It divides the CPU pipeline into the frontend and backend, constructs the instruction execution graph, and uses performance counters to locate the pipeline bottleneck hierarchically. Though effective, these tools focus on on-chip profiling and cannot associate core-level inefficiencies with off-chip CXL memory access. Regarding memory subsystems, existing tools [13, 43, 80] examine the latency-bandwidth curve under different memory loads and access patterns. They report how workloads use CXL DIMMs from the performance aspect, but fail to diagnose the pathological scenarios caused by a CXL flow, such as stalling the processor, causing head-of-line blocking to local memory requests, or squandering the CXL link bandwidth. In terms of interconnect or system bus, there are benchmarking frameworks [55, 69, 86] to characterize intra-host ongoing NoC events and PCIe transactions, but they barely capture how these data movements affect the source (core) and destination (memory). **Thus, we lack a systematic and end-to-end profiling tool to understand and analyze CXL.mem-based systems and applications.**

3 Dissecting CXL.mem Execution

We delve into the processor performance monitor unit (PMU), explore a variety of counters, and use them to characterize the execution of different CXL.mem paths (§2.2).

3.1 PMU Overview

A PMU is a specialized hardware module that monitors the processor execution characteristics. It exposes a programmable interface for developers to select and configure monitored event registers, and generates running statistics for people to tune and optimize system software, compiler, and application performance. Generally, there are three categories of performance counters. The first one captures processor events, counting the frequency that a prescribed event has occurred at which micro-architecture components during the profiling period, such as issued/retired instructions, cache/buffer hit/miss, coherent messages, etc. The second one focuses on core execution and measures the pipeline stall cycles under what conditions, for example, the buffer full, credit starvation, or incomplete

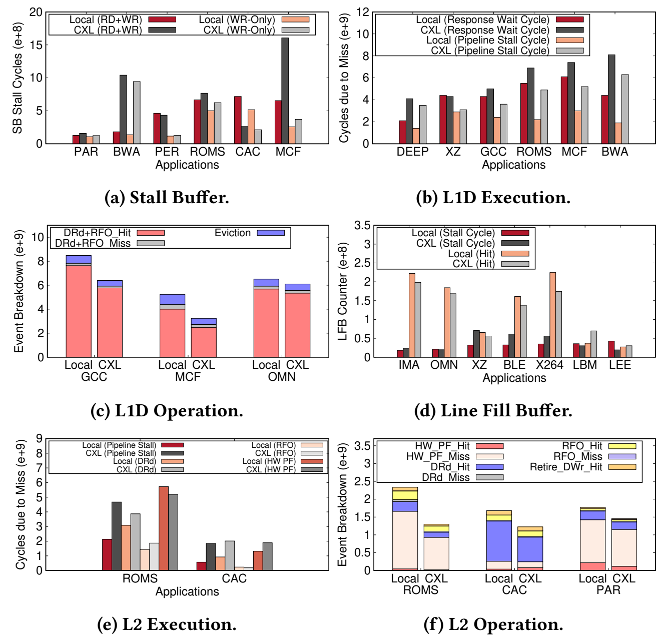

(a) Stall Buffer. (b) L1D Execution. (c) L1D Operation. (d) Line Fill Buffer. (e) L2 Execution. (f) L2 Operation.

Figure 2: We compare core performance counters when running in the local and CXL memory cases. §5.1 describes our experimental setup. (a) reports the core stall cycles of six applications when the store buffer becomes full under the read/write mixed and write-only cases. (b) and (c) present the L1D characteristics from the execution and operation perspectives. (d) shows the LFB counter statistics. (e) present the core stall cycles and data responses under L2 misses, while (f) depicts the L2 operation breakdown.

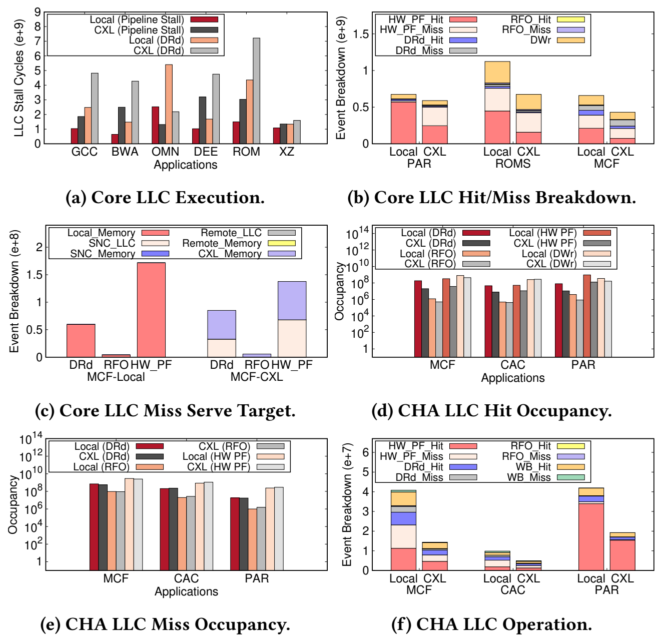

(a) Core LLC Execution. (b) Core LLC Hit/Miss Breakdown. (c) Core LLC Miss Serve Target. (d) CHA LLC Hit Occupancy. (e) CHA LLC Miss Occupancy. (f) CHA LLC Operation.

Figure 3: We compare CHA PMU when running in the local and CXL memory. (a) reports the core LLC stall cycles. (b) presents the LLC hit/miss breakdown. (c) shows where the missed LLC requests are served. (d)/(e) report the occupancy of LLC hits and misses. (f) depicts the LLC operation breakdown. The y-axis of (d)/(e) is log-scale.

read. The third one centers around the data feeding pipeline, counting the time it takes to fetch a data response from the destination to the current position, e.g., demand read response waiting cycles at L1D. A PMU counter usually operates in two modes: (a) continuously profiling, reporting the total amount until receiving the stop/reset signal; (b) sampling, firing an overflow interrupt when the counter reaches a predefined threshold.

Based on our `CXL.mem` data path analysis, we divide PMUs into four parts, i.e., core, CHA/LLC, Uncore, and CXL device. We compare applications running atop local versus CXL memory and see how the performance counters are affected. When drilling down the results, our goals are (a) examining microarchitectural pipeline behaviors in the case of slow CXL memory accesses; (b) identifying and mapping performance counters to different data paths; and (c) understanding the counter capabilities and limitations.

3.2 Core PMU

The core PMU can capture the origins of four `CXL.mem` data paths and indicate how load/store instructions are issued. We further break it down into SB, LFB, L1D, and L2.

Stall Buffer. It has two counters (Table 1 in Appendix), measuring stalled cycles due to SB full under read/write mixed and write-only (no ongoing loads) scenarios. Using these two, one could quantify the impact of write intensity at the individual core, i.e., how exorbitant writes block the pipeline. As shown in Figure 2, compared with local memory, accessing the slow CXL DIMM causes stalled cycles to increase by 1.9× and 2.0× on average across six applications in the RD+WR and WR-only cases, respectively. This is because store-induced cache writebacks take longer to commit.

L1D Cache. The core PMU introduces 12 performance counters for L1D. We chose five representative ones (Table 1): three counting L1D hits, misses, and evictions, and the rest measuring execution stall cycles and data response time under L1D miss. This allows us to analyze execution statistics, understand the L1D locality, and track DRd, RFO, and SW PF paths. As depicted in Figure 2-b, when running applications over CXL memory, we observe 2.1× higher pipeline stall cycles on average than the local one, along with 1.4× longer response wait time. Even though workloads in both local or CXL memory scenarios issue the same amount of loads and stores, as CXL slows down the pipeline, data locality is inherently affected. As shown in Figure 2-c, on average across three workloads, 22.8% fewer DRd and RFO hits happen under CXL, indicating that cached entries benefit little for future requests, and the L1D efficiency is reduced. Note that under an L1D miss, a cache entry is filled or replaced depending on the cache replacement policy and how multi-level caches are structured (i.e., inclusive or exclusive).

Line Fill Buffer. There are two LFB-related counters (Table 1) that report the number of demand load hits and the number of stall cycles when LFB is full (unavailable). Akin to SB, they characterize the load intensity of DRd and RFO paths after missing from L1D. As shown in Figure 2-d, CXL memory changes the data locality. Among 7 applications, 519.lbm_r and 541.leela_r see an 88.5% and 12.0% hit increase, yielding 15.4% and 54.6% stall reduced, respectively. However, the rest experience a 14.2% hit decrease and a 59.2% more stall cycles. The longer data response under CXL would benefit applications whose temporal access locality is a little farther apart, i.e., long data reused distance [41].

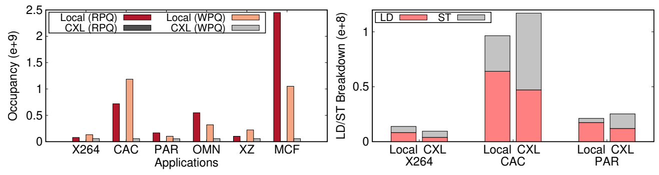
L2 Cache. The core PMU provides 44 performance counters, and we select 25 for analysis (Table 1). Among them, there are (1) 18 event counters to track the hit and miss information of retired and speculative instructions for DRd, RFO, DWr, and HW PF paths; (2) 7 cycle counters measuring the pipeline stall and data response time of each path. Software prefetching (SW PF) is merged into the DRd path after missing from L1D. As shown in Figures 2-e/f, using these counters, we can learn L2-induced pipeline stalls due to memory loads/stores, their data response time, and the frequency of different L2 events. For example, compared with local memory, we observe the core stall cycles are increased by 2.7× on average when running over CXL memory, along with 1.2×, 1.7×, 1.1× longer data response for HW PF, DRd, and RFO requests. However, similar to what we discussed before, the data locality is changed in the CXL case, indicating that some data paths might become faster, e.g., 554.roms_r and 507.cactuBSSN_r achieving 9.4% and 20.5% lower data feed waiting cycles on the HW PF and RFO paths, respectively. Further, comparing CXL and local memory, we find that the number of HW PF hits, HW PF misses, DRd hits, DRd misses, RFO hits, RFO misses, and DWr hits is reduced by 0.7%, 27.5%, 33.3%, 11.5%, 18.4%, 0.6%, and 25.8% on average, respectively. Such a trend is not consistent across all applications. For example, 507.cactuBSSN_r sees a 2.0× and 1.5× increase in HW PF hit and RFO miss. More speculative load/store-related instructions (i.e., 46.3% on average) are issued in the CXL case due to longer execution. Thus, with these counters, one can monitor the per-core L2 data locality and zoom in on the DRd, RFO, and HW PF request execution.

3.3 CHA PMU

The CHA maintains cache coherence and provides LLC monitoring counters at the core/socket level (Table 2 in Appendix).

Core LLC. The core-level CHA provides 81 counters and we choose 60. Specifically, there are (a) 2 cycle counters measuring the DRd-induced core stall and data response; (b) 10 event counters monitoring cache coherence; (c) 48 counters reporting the access hit and miss for DRd, RFO, and L1D/L2 HW PF paths. CHA offers adequate counters to demystify the traffic load of different paths. For example, under an LLC hit, based on the HitM bitmap, it can show if the data is served from the local-core LLC slice, cross-core local-chiplet LLCs, cross-chiplet LLCs, or a snoop response. For an LLC miss, it tracks where the data is served, e.g,. local DRAM, cross-chiplet DRAM, remote (cross-socket) cache, and remote/CXL DRAM. The CHA further introduces an occupancy counter for a request type, measuring how many such requests are missed per CPU cycle, indicating the missing degree.

We observe that accessing CXL memory causes 2.1× more stall cycles and 1.8× higher DRd response time on average across six applications when compared with the local case (Figure 3-a). Memory access pattern also changes in the CXL case. As shown in Figure 3-b, the number of LLC hits is reduced by 46.5%, 41.3%, and 62.2% across the DRd, RFO, and HW PF paths on average, while the total LLC misses is raised by 4.2×, 4.0×, and 5.3×, respectively. The missed LLC requests are served from six locations (as discussed above). In the non-CXL case, most DRds, RFOs, and HW PFs (more than 99.0%) are served from the local DIMM. However, when running under CXL, we find that 38.4%, 4.1%, and 49.2% of the misses can be



(a) Queueing Occupancy. **(b) Load/Store Breakdown.**

Figure 4: We compare Uncore PMU of applications running in the local and CXL cases. (a) reports the channel occupancy of RPQ and WPQ. (b) breaks down the load and store commands.

directly served by the cross-chiplet/socket remote cache via snooping, then followed by the CXL DIMM. This indicates that longer memory access latency increases the core LLC cache locality.

Socket LLC and CCD Counters. The CHA PMU provides 686 counters at the socket/CCD level to monitor how LLC slices serve requests from all cores. We choose 108 counters, including 39 event and 69 cycle counters to track (a) LLC cache coherence state machines (like E→F/I, M→E/I, and S→I); (b) cross-core snoop filters; (c) hit, miss, and occupancies across DRd, RFO, and HW PF paths; and (d) the destination distribution of socket-level LLC misses. As shown in Figures 3-d/e, compared to the local memory case, accessing CXL memory reduces the hit occupancy of DRd, RFO, HW PF, and DWr on average by 86.3%, 50.0%, 84.6%, and 29.5%. Concomitantly, the miss occupancy is increased by 4.8×, 1.3×, 1.1×, and 3.0×. This is due to the longer uncore data path. We further break down the traffic load of DRd, RFO, HW PF, and DWr paths (Figure 3-f) and find that their hits under CXL memory are reduced by 55.4%, 48.0%, 59.4%, and 44.2% compared to accessing local memory.

3.4 Uncore PMU

The Uncore PMU captures the mesh interconnect and FlexBus I/O behaviors (Table 3 in Appendix). We divide it into two parts: IMC (integrated memory controller) and M2PCIe. The IMC also participates in the CXL context, serving as a host-side agent to streamline CXL memory transactions [84, 85].

IMC. It provides 54 performance counters, among which we select 18 for data and write paths analysis. An IMC monitors the memory module and exposes two virtual channels, where each features 9 counters. Specifically, there are (a) 4 cycle counters tracking the number of cycles and occupancy when the Read Pending Queue (RPQ) and Write Pending Queue (WPQ) are not empty; (b) 3 event counters to capture CAS (column address strobe) commands; and (c) 2 counters tracking the queue slot allocation in RPQ and WPQ. We notice that in the case of CXL, because the CXL DIMM also encloses device-side command queues, unlike the local memory, little queueing happens inside the IMC (Figure 4-a). Thus, one can ignore the IMC impact when analyzing CXL-only memory streams. This also indicates that under mixed memory streams, the local DIMM-induced queue at IMC could potentially block CXL accesses.

M2PCIe. It provides 34 performance counters to track FlexBus I/O, from which we select 4 counters to monitor the CXL load/store transactions. Among them, two report the number of entries inserted into the M2PCIe ingress queue from the mesh and the number of cycles during which the ingress queue is non-empty. The rest two counters track the number of CXL memory acknowledgments sent from the M2PCIe egress queue to the mesh and the number of cache line data transferred. These counters are per endpoint (per DIMM), showing the actual CXL traffic. Figure 4-b compares the load/store in the local DIMM (using IMC) and CXL DIMM (using M2PCIe). The total memory accesses are roughly the same. However, since CXL accesses are slow, the average load/store amount in the same profiling cycle is 36.7% lower compared with the local scenario.

3.5 CXL PMU

The CXL Type-3 device defines 56 counters (Table 4 in Appendix), where 8 are used for performance diagnostics of M2S Red/RwD and S2M DRs/NDR (§2.1). These include (a) 6 counters to monitor the ingress packing buffers, i.e., tracking the number of inserts, the number of cycles during which the buffers are non-empty, and the number of cycles when the buffers are full for both the Mem Data packing buffer and the Mem Request packing buffer; (b) 2 counters track the number of inserts in the Mem Data egress packing buffer and the Mem Request egress packing buffer. These are used to derive the QoS telemetry for memory (i.e., light load, optical load, moderate overload, and severe overload), officially introduced in the CXL specification 3.0/3.1 [18]. However, existing CXL DIMMs (like AMD CZ120/CZ122, Smart Modular CMM-E3S, and Samsung CMM-B/CMM-H) are not supported. We'll explore it in the future.

3.6 PMU Generality

The above characterizations (§3.2–§3.4) are performed on an Intel SPR machine. Next, we show that these PMUs are generally available and applicable to another Intel EMR server.
Core PMU. We observe consistent core PMU counter behaviors when accessing CXL memory. As shown in Figure 14 (Appendix A.1), compared to local memory, accessing the slower CXL DIMM (a) results in 1.3× more SB stalled cycles on average across six applications in both RD+WR and WR-only scenarios; (b) leads to an average of 1.3× higher pipeline stall cycles and 1.2× longer response wait times on L1D; (c) causes 2.7% fewer DRd and RFO hits on L1D; (d) increases the LFB stall cycles by 1.3× with little changes in LFB hit rate; (e) yields 1.5× more L2 stall cycles on average, with HWPF, DRd, and RFO requests experiencing 1.1×, 1.3×, and 1.2× longer data response times; (f) reduces RFO misses, HWPF hits, and HWPF misses by 10.8%, 1.5%, and 4.3%. The EMR machine exhibits smaller increases in stall cycles, and less variation in hit and miss counts, benefiting from the larger LLC size, but basically shows the same trend as SPR when accessing CXL memory, with an increase in stall cycles and a decrease in request frequency.
CHA PMU. These counters are also applicable on the EMR server. As shown in Figure 15 (Appendix A.1), compared to local memory, accessing CXL memory (a) leads to 2.1× more stall cycles and 2.2× higher DRd response time on LLC on average across six applications compared to the local memory case; (b) decreases the number of LLC hits for DRd, RFO, and HWPF requests by 6.3%, 16.3%, and 5.6% on average; (c) causes a 1.1× and 1.3× more LLC misses for DRd and RFO, while HWPF misses are dropped by 20.5%; (d) increases the DRd, RFO, and HWPF miss occupancy by 1.5×, 1.3×, and 1.6×; (e) yields a 2.2%, 1.8%, 3.6%, and 4.3% hit decrease for DRd, RFO, HWPF, and DWr, along with a 2.3%, 6.8%, 5.2%, and 3.4% miss reduction.
Uncore PMU. We also validate that the functionality and effectiveness of the uncore PMU on an EMR server are similar to that of the SPR one. Figure 16-a (Appendix A.1) is consistent with Figure 4-a, confirming that the CXL memory access bypasses IMC and is managed by a device-side MC. Akin to Figure 4-b, Figure 16-b compares the load and store traffic between local and CXL DIMMs, providing DIMM traffic ground truth information.

By comparing PMU counters when accessing local and CXL memory on both the SPR and EMR machines, we demonstrate that our PMU-based characterizations and experimental observations are generally applicable. Further, a large LLC on the EMR platform helps reduce the core stall cycles and L1D/L2/LLC misses.

3.7 Summary

PMU counters allow us to (a) investigate the impact of slow CXL memory accesses on microarchitectural pipeline behaviors; (b) classify and quantify different `CXL.mem` data paths. Among these PMUs, CHA, M2PCIe, and CXL PMUs offer ground-truth information on CXL read/write memory traffic and capture its effects at the uncore level. Core PMUs enable analyzing CXL memory characteristics of different paths (i.e., DRd, DWr, RFO, and HW/SW PF) from the uncore and core private cache to the processor pipeline.

4 PathFinder: a `CXL.mem` Profiler

Our characterizations (§3) have unearthed the PMU capabilities of different hardware components across `CXL.mem` data paths. This section describes how we build PathFinder using them to address the above challenges (§2.3). Our system design goals are:

- **End-to-end.** PathFinder should provide adequate running statistics about all application-induced `CXL.mem` data paths and analyze how CXL memory accesses impact the processor pipeline, cache hierarchy, and FlexBus I/O;
- **Profiling-rich.** PathFinder should report multifaceted execution telemetry, including stall cycle, queueing occupancy, and data locality, at a given epoch, based on the profiling specification;
- **Lightweight.** `CXL.mem` transactions run at nanosecond granularity across tens of GB to several TB address spaces. It is pivotal to ensure that PathFinder incurs minimal system overheads with a marginal impact on the profiled applications;

4.1 Key Idea and System Overview

Our key idea is to view the server processor and its chipset as a multi-stage Clos network [30, 38], identify different `CXL.mem` data paths, and apply conventional traffic analysis techniques. In the request direction, the ingress stage is processor cores that feed load/store instructions into the network, while the egress stage is CXL DIMMs that serve requests and perform data reading and writing. The response direction operates vice versa. The middle stage is each on-path architecture module, working as a switch that forwards data based on memory addresses. However, in our case, a middle stage can also become an ingress or egress one when hardware prefetching or caching happens. This indicates that our Clos network has several sub-Clos networks with non-consistent stages for different `CXL.mem` paths. Next, we develop an in-band traffic telemetry engine for each stage hop to capture aggregated running statistics using its PMU. Last, we employ networking telemetry techniques, like traceroute [23], reverse traceroute [60], delay-based analyses [32, 36, 44], and network snapshot [103], and tailor them to the `CXL.mem` context to achieve our goals.

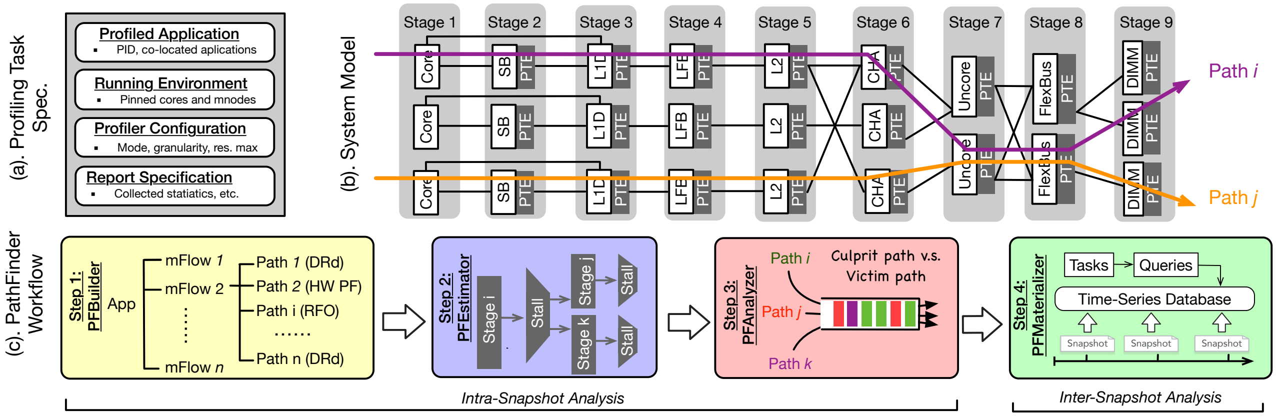


Figure 5: The system overview of PathFinder. (a) shows the profiling task specification. (b) depicts the system model, and (c) illustrates the PathFinder workflow. PTE=PMU-based Telemetry Engine.

PathFinder is a system utility facilitating CXL development. Its inputs (Figure 5-a) encompass (1) application programs, where we support single/multi-tenant profiling; (2) running environments, such as pinned cores and mapped memory nodes; (3) profiler specifications, which set up the profiling mode (continuous or aggregated), tracing granularity, and the max resource consumption; (4) report specifications, which describe the interested execution statistics. Based on these configurations, PathFinder outputs profiling summaries periodically or until the application stops, and highlights notable `CXL.mem` information, such as what is the access intensity across different CXL paths, how the data locality varies under CXL accesses, what is the latency breakdown of load/store/prefetch, whether there exists a hardware bottleneck blocking some paths, and how CXL memory flows interfere with other local/CXL ones.

Under the hood, PathFinder performs snapshot-based path-driven profilings to analyze `CXL.mem` protocol execution (Figure 5-c). It takes time series snapshots over our telemetry engines, classifies transactions based on paths, and examines how concurrent paths interleave over each on-path module. PathFinder comprises four techniques: (a) *PFBuilder*, constructing the CXL data path map via carefully synthesizing a slew of PMU counters (§4.3); (b) *PFEstimator*, which employs a back-propagation algorithm that gradually attributes the CXL-induced stall cycles in a bottom-up fashion to the CPU pipeline (§4.4); (c) *PFAnalyzer*, zooming each architectural component via white-box modeling and exploring how concurrent CXL and non-CXL streams interfere with each other (§4.5); (d) *PFMaterializer*, which introduces an internal time-series database, takes a per-snapshot digest as inputs, and identifies consistent execution characteristics, such as data locality, contention, and resource underutilization ($4.6). Together, PathFinder dissects `CXL.mem` accesses and associated applications across the entire server system from the temporal and spatial dimensions at the required granularity.

4.2 System Model

PathFinder models the server system as a Clos network (Figure 5-b), represented as $G = (V, E)$, where (a) V refers to architectural modules, like core, SB, LFB, L1D, L2, CHA, and CXL DIMM; (b) E is the interconnect link, such as on-core hardware FIFO, (mesh) interconnect, and system FlexIO Bus. Each vertex can be a source node (v_{src}) issuing requests, a destination node (v_{dst}) serving requests, and an intermediate stage (v_{int}) with M ingress and N egress ports, which are all associated with a particular stage (S). We then define a local/CXL memory flow (*mFlow*) as $Core_i \leftrightarrow lDIMM_j/cDIMM_j$ (Figure 5-a), enclosing all the associated load, store, and prefetch commands and responses following the committed order. A mFlow is (1) *application-dependent*, whose lifetime aligns with the workload; (2) *location-sensitive*, i.e., we would create and initiate a new mFlow when the thread migrates to a new core or touches the address space of a new DIMM; (3) *bidirectional*, capturing both the request submission and response returning transmission. Thus, an application process would include a list of mFlows $\{mFlow_1, ..., mFlow_n\}$, whose number is bounded by $Core\# \times DIMM\#$.

A mFlow spawns a list of data paths (P) based on the data serving location, each of which is denoted as $\bigvee v_i \overset{e_k}{\longleftrightarrow} v_j$, where $v_i, v_j \in V$ and $e_k \in E$. A path is instantiated when a load/store instruction is issued. It is deterministic based on the address mapping, whose forward and backward sub-paths are symmetric. PathFinder uses the *PFBuilder* (§4.3) to break down path details of all active mFlows. It takes a snapshot of all PMUs at the end of every OS scheduling epoch (or when preemption happens) and associates it with the current running mFlow. We then generate an execution digest, represented with a memory-efficient data structure. Thus, a mFlow has a sequence of time-series snapshots $\{Snapshot_1, ..., Snapshot_n\}$ over its lifetime, facilitating our profiling over the time dimension.

4.3 Constructing the Path Map

Our first step is to draw all the paths over each snapshot. *Traceroute* is a widely used technique in computer networks to determine which routers stay on the communication path. The client sends a probing packet with a predefined TTL (time-to-live) value. Each on-path router receiving the probing decrements the field and replies to an ICMP time-exceeded message [11] when TTL becomes 0. One would then figure out the packet path by gradually incrementing the TTL field until reaching the destination. However, such a tracing scheme is infeasible in our case due to the non-programmability of on-path micro-architecture hardware components. Load/store requests and responses usually traverse a deterministic path, instructed by the memory address. One possible solution is to analyze the destination address region and work with the server vendor to demystify the caching policy, multi-tier cache structure, and

NoC (Network-on-Chip) routing algorithm. For example, the recent 432-core RISC-V server processor Occamy [87] open-sources its architectural details. This would not work for commodity server processors. Fortunately, we find that PMUs report path-specific hit-and-miss information (§3) at different locations, allowing us to reconstruct the path map within a snapshot.

We develop the *PFBuilder* mechanism guided by Table 5 (Appendix A.2). For each mFlow, it first examines its home core PMU and computes the per-path traffic load based on how many DRd/RFO/PF/DWr hits happen at SB, L1D, LFB, and L2. Next, it walks through the PMU hierarchy in a top-down fashion and analyzes the CHA PMU. Since each core can send load/store commands to all CHAs, the next hop after L2 becomes unclear, hinging on the propriety routing logic, LLC slice caching, and destination address. We find a special hardware module–called TOR (Table of Requests) in the Intel processor, which records the core-CCA mapping for different types of requests (see `unc_cha_tor_inserts` in Table 5). This helps us continue building the path map for missing commands from L2. The AMD Zen processor has a functionally similar component in its core complex die (CCD). Note that LLC request serving has several scenarios. Specifically, a miss would be first served by (a) the core's local LLC slice, (b) followed by a distance LLC slice from a Sub-NUMA cluster (when a processor has chiplets), and (c) finally followed by a remote LLC slice from another CPU socket. (b) and (c) are triggered by request snooping. Finally, if requests were missed from LLC/CHA, they would go through MC and arrive at the DIMM. We use local MC counters at the socket level for local memory access and M2PCIe uncore counters to track the FlexBus transactions to CXL DIMMs. In sum, a path map describes all the mFlow-induced paths with quantitative traffic loads (defined as the number of traversed requests). A snapshot, taken at the end of a scheduling window, is always associated with a particular thread. However, when there are multiple concurrent flows (e.g., a core accessing different DIMMs), PathFinder can classify paths whose destinations are beyond LLC since their PMUs can report target-dependent statistics (as shown in Table 5), but not the on-core ones as all path hits are mingled in the counter.

4.4 Breaking down CXL-Induced Pipeline Stall

The longer CXL accesses slow down data fetching, inevitably causing execution stall along the pipeline. PathFinder examines the stall cycles of every hardware component and aims to tease out the CXL-induced part. The challenge is how to separate the combined effect between CXL transactions and other architectural factors (like local memory access and cache coherence execution). Inspired by reverse traceroute [60] from the computer networks, we develop a back-propagation approach (*PFEstimator*) that incrementally attributes the stall overhead from CXL DIMMs to cores.

PFEstimator works as follows (ALG 2 in Appendix A.3). Starting from the CXL_DIMM MC, it checks which FlexBus root access ports (RCs) have issued requests in the current snapshot, computes their aggregated traffic loads of all active paths, and proportionally distributes the load/store queueing occupancy (obtained from the `unc_cxlcm_rxc_pack_buf` counters) to these RCs (L2-L9). The queue buildup at a CXL DIMM MC happens because its memory command handling rate (egress) cannot catch up with the request arrival rate (ingress). Next, we move backward to the

Algorithm 1 The Delay-based PFAnalyzer Algorithm.

```
 1: HAL: the hardware abstract representation that contains its PMU
    counter statistics and all housed path/mFlow information;
 2: Q: queue length array;
 3: procedure HW_OCCUPANCY_DETECTION(HAL module)
 4:     for p in module.all_cxl_paths do
 5:         for c in module.all_components do
 6:             if c == L2 or c == L1D or c == LLC then
 7:                 $\lambda_{hit}$ = c.hits[p]/c.clocks;
 8:                 $\lambda_{miss}$ = c.misses[p]/c.clocks;
 9:                 $W_{hit}$ = c.delay_hit;
10:                 if c == LLC then
11:                     $W_{miss}$ = c.delay_miss;                    ▷ LLC
12:                 else
13:                     $W_{miss}$ = c.delay_tag;           ▷ core L2,L1D
14:                 Q[p][c]=$\lambda_{hit}$*$W_{hit}$+$\lambda_{miss}$*$W_{miss}$;
15:             else                                    ▷ core LFB,RAM
16:                 $\lambda_{hit}$ = c.hits[p]/c.clocks;
17:                 $W_{hit}$ = c.delay_hit;
18:                 Q[p][c]=$\lambda_{hit}$*$W_{hit}$;
19:     culprit_path = MAX_OCC(Q);
```

FlexBus RC→Host Uncore segment and divide the FlexBus waiting (credit starvation) cycles–`unc_m2p_rxc_cycles_ne`). The estimation works similarly (L10-L18) except that the stall cycle needs to consider the inherited part from the last segment (L12). After that, *PFEstimator* examines the Host Uncore→CHA path segment and proportionally attributes its RPQ (read pending queue) and WRQ (write pending queue) delay (i.e., `unc_m_rpq_cycles_ne` and `unc_m_wpq_cycles_ne`) to different CHAs (L19-L27). Since the uncore is shared between local and CXL memory streams, `get_Uncore_stall_cycle` takes the CXL DIMM ID as input and reports the DIMM-specific value (L21). As discussed in §2.2, within a CHA, there is a mapping between LLC slices and CCDs. We further partition the stalled cycles (observed via TOR) proportional to individual slices. This works similarly to Host Uncore→CHA. Finally, for all the in-core path segments (Core LLC → L2 → LFB → L1D → SB), the CXL-induced stall cycles are back-propagated to each on-path module as above while piggybacking a proportional stall cycle share (like L21). Note that algorithm 2 sketches a generic description and the actual implementation would use different counters for load (read) and store (write) requests.

4.5 Detecting Culprit Paths at Bottlenecked HW

Next, PathFinder locates the hardware contention point along the pipeline, explores the performance interference of concurrent paths, and identifies the culprit. However, due to the hardware opaqueness, it is challenging to know how different mFlows interact with each other and share the hardware resources. Delay-based queueing analysis [32, 36, 44] has been demonstrated in computer networks. The idea is to attribute the queueing occupancy of a flow to the individual flow based on its delay variation. We introduce a similar approach (called *PFAnalyzer*), which leverages Little's Law and the delays observed by the core when requesting data from different components to attribute the queueing degree of flows.

PathFinder views each hardware (vertex) in the Clos network graph as a software switch and models it as a queueing model. An observation driving our delay-based analysis is that the PMU of an architectural module generally encloses two counters (§3): (a) one counting the hit and miss frequency (*HitCnt*,*MissCnt*), which relates to the data locality and total amount of allocated resource from the last epoch; (b) the other reporting the data response time (*Delay*), which indicates how long the requests would be served. Except for interconnect routing, components along the data path can be modeled as a variant of the FCFS queue (S3-FIFO [102]). *PFAnalyzer* combines the *HitCnt*, *MissCnt*, and *Delay*, and applies Little's Law to estimate the average queue length per cycle as $L = \lambda * W$. We attribute the core-observed request latency to each on-path component by computing the latency difference between the current hop and the previous hop, used as the delay W.

For components that forward miss requests to lower levels, we use the following extended formulation, $L = \lambda_{hit} * W_{hit} + \lambda_{miss} * W_{miss}$, where (a) λ denotes the request arrival rate, obtained from the component's mFlow hit/miss counters and clock ticks; (b) W_{hit} represents the time required to serve data on a hit. W_{miss} is location-dependent. For L1D and L2, we use W_{tag} as W_{miss}, which captures the time spent on tag lookup to determine whether data exists. W_{tag} is assigned a constant cycle value based on the hardware capacity and associativity. For LLC, we use the request miss delay as W_{miss}, since missing requests remain in the CHA TOR queue until they are completed. For LFB and DIMM, as the LFB load is part of the uncore path [98], and memory holds the complete data set that is no longer forwarded, we adopt the $L = \lambda_{hit} * W_{hit}$ model. Algorithm 1 sketches how *PFAnalyzer* works. For DRd, RFO, and HWPF paths, we apply different models to estimate the queueing degree at each component along the path. At L1D, L2, and LLC, both hit and miss accesses are included in the queue length estimation (L6-L14). At LFB and DIMM, only hit accesses are considered (L15–L18). Finally, the component and path with the maximum queue length are identified as the culprit of the current snapshot (L19).

4.6 Synthesizing Multi-Snapshots

PathFinder summarizes execution characteristics via cross-snapshot analysis. We employ a time-series database (like InfluxDB [5]), encapsulate a snapshot as a compacted record, and conduct time-series analysis. Our record, tagged by the timestamp, uses a hierarchical tree representation based on the system model (§4.2), including edges, vertices, mFlows, and paths tables. An edge is further divided into the traversed path list (§4.3), traffic load, available bandwidth, and queueing degree (§4.4). A vertex adopts an abstract hardware model, contains its PMU counters, and captures how resources are allocated among contending paths (§4.5). A mFlow/path encompasses basic routing information and running metadata statistics.

Our approach (*PFMaterializer*) provides a CLI interface, takes a user-interested scenario as input, and translates it into a sequence of InfluxDB Flux queries to explore insights. Supposing we are analyzing the application's LLC temporal data locality when accessing CXL memory. First, *PFMaterializer* determines the query scope that contains the application-spawned paths whose destination is LLC, i.e., `FROM "path_set" WHERE "path.mflow.pid = APP_PID" AND "path.dst=LLC"`. Second, we look at the counter hit field and get some overall statistics via some operators, such as `min()`, `max()`, `avg()`, and `movingAverage()`. Third, *PFMaterializer* employs the time series cluster technique [29] and partitions snapshots into multiple windows with similar hits. The window length reflects how long an application stays consistently in the current phase. Fourth, *PFMaterializer* applies classical time series analysis (TSA) techniques [53] to explore data trend, seasonality, and residual (or anomaly). For example, it can use the Holt-Winters forecast method–`holtWinters()`–to search regular patterns, which would indicate if an application exhibits some predictable data accesses. Last, thanks to the time-series database capability, we can cross-check other applications' mFlows/paths in the same time window or at the same timestamp, use some correlation detection methods, like `pearsonr()`, and identify locality-impacting factors from the application layer. This workflow is generally applicable and can be easily extended via other queries. By layering a time-series database atop the PathFinder system core, we enable many other architectural-level CXL-related performance profiling, like spatial data locality, computing burst, and execution orthogonality. We will open-source and continuously develop more workflows.

5 Evaluation

5.1 Experimental Methodology

Hardware testbed. We conduct experiments on two types of hardware platforms. The first is a dual-socket Sapphire Rapids (SPR) server with Sub-NUMA Clustering (SNC) enabled. It is equipped with one CXL Type-3 memory device that appears as a CPU-less NUMA node and runs on Linux 6.5 kernel with CHA PMU support patches. Our experiments run on a 2U Supermicro server that has two Intel Xeon Gold 6438Y+ processors and 256GB DDR5. The CPU has 32 cores running at 2.0 GHz and 60MB LLC. Each core has 48KB L1D and 2MB L2. We disable hyperthreading and Turbo Boost. Our CXL Type-3 memory device is based on the Intel Agilex I-Series card [6] enclosing 16GB DDR4. The second platform is a dual-socket Emerald Rapids (EMR) server. It uses 256GB CXL Type-3 memory device Micron CZ120 CXL DIMMs configured as CPU-less NUMA node and runs on Linux 6.15 kernel. The server features two Intel Xeon Gold 6530 processors and 1536GB DDR5 memory. The CPU consists of 32 cores with 160MB LLC, and each core includes 2MB L2 and 48KB L1D cache. Both servers are equipped with per-core PMUs, 64 CHA PMUs, and use IMC and M2PCIe PMUs to monitor local and CXL memory accesses, respectively.

Workload. We evaluate PathFinder using 77 applications from various benchmark suites, including Redis [14] and YCSB [39], graph processing GAP [33], PARSEC [35], SPLASH-2x [105], and SPEC CPU 2017 [17]. PathFinder provides a CLI interface with different command parameters to enable various functionalities.

5.2 Case 1: Path Classification

PFBuilder uses the PMU hit/miss counters to figure out the precise request paths, reporting the total amount and distribution of request hits and misses at each cache hierarchy and DIMM component from a core to the destination. This shows detailed memory access behaviors and traffic changes along different paths.

Table 7 (Appendix A.5) demonstrates PFBuilder's path mapping capabilities when running SPEC CPU2017 applications over CXL memory. It reports the path distribution for DRd, RFO, HWPF, and DWr. For example, in 649.$fotonik3d\_s$, the per-core hot path is DRd. At the uncore, the hot path is HWPF, accounting for 59.3% of uncore accesses. Further, PFBuilder shows that the CXL memory hits are 8.1× more than the local LLC hits, and HWPF paths account for

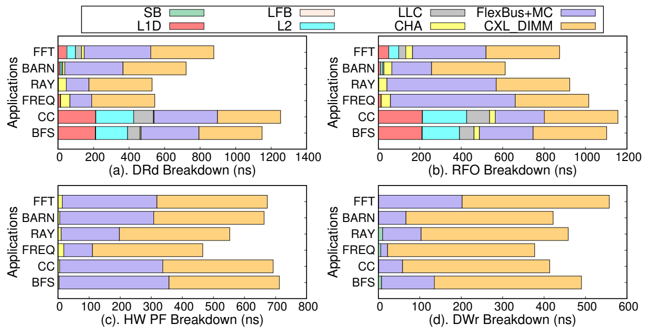


Figure 6: We break down CXL-induced stall cycles of SB, L1D, LFB, L2, CHA/LLC, FlexBus+MC, and CXL DIMM.

89.1% of those CXL memory accesses, suggesting that CXL memory heavily influences L1D and L2 behavior through the HWPF path. As another example, we compare two selected snapshots in 602.*gcc_s*, and find out that the total number of core-issued requests (from DRd, RFO, and DWr hits across all components) increases by 5.8× in snapshot 2 compared to snapshot 1. PFBuilder allows us to drill down into the CXL memory hit analysis: (a). The total amount of DRd requests only slightly increases from 25.9% to 27.7%, indicating that the cache hierarchy largely absorbs the additional DRd traffic; (b). In contrast, RFO requests rise from 1.1% to 69.0%, meaning that a large amount of data is loaded from the CXL memory.

5.3 Case 2: Pipeline Stall Breakdown

PathFinder can examine the CXL-induced stall cycles and provide a breakdown through *PFEstimator* (§4.4). This functionality allows developers to understand (a) how efficient the CXL accesses are from the processor pipeline perspective; (b) how many stall cycles are added from SB to FlexBus+MC. However, in a mixed memory traffic scenario [79, 97], PMU stall cycle counters capture the combined impact of both local and CXL memory paths. Separating stalls based solely on the proportion of request miss targets is inaccurate [95]. PFEstimator adopts a bottom-up back-propagation approach to tease out the CXL-induced portion from the total stall counters and presents a stall breakdown to unearth the extent each component is affected by CXL memory.

Figure 6 reports the breakdown for six applications. Regarding *fft*, its stall delay of the DRd path is distributed across 7 components (L1D, LFB, L2, LLC, CHA, FlexBus+MC, and CXL DIMM) with a percentage of 5.7%, 0.0%, 5.5%, 3.9%, 1.7%, 42.7%, and 40.3%, indicating that DRd execution stalls are more pronounced in the uncore. However, *raytrace* experiences a higher stall at FlexBus and MC with 67.1%. Regarding RFO, we observe a breakdown of 1.9%, 0.0%, 1.6%, 0.9%, 6.3%, 3.3%, 58.0% and 1.3%, 0.01%, 0.07%, 0.03%, 4.3%, 59.3%, 35.0% stalls happen for *barnes* and *freqmine* applications across the data path. The hardware prefetch also causes a stall. For example, we find that 45.2% and 52.7% stall cycles come from FlexBus+MC and CXL DIMM for the *fft* application, similar to the DWr paths.

Next, we show the advantage of capturing the CXL memory impact through bottom-up back-propagation. In a hierarchical cache system, CXL-induced stalls gradually diminish from the uncore toward the core due to locality. For example, in Figure 6-a, CXL-induced stalls on the DRd and RFO paths decrease by an average of 74.5% and 67.8% from FlexBus+MC to L1D. The PFEstimator's

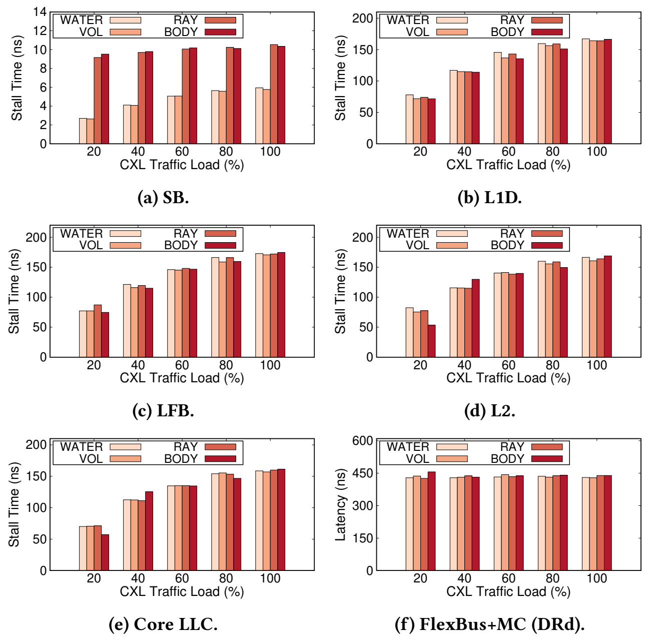
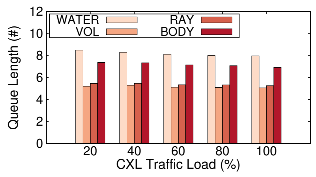


Figure 7: We present the pipeline stall cycle as varying the CXL traffic load. (a), (b), (c), (d), (e), and (f) report the behavior for SB, L1D, LFB, L2, core LLC, and FlexBus+MC, respectively.

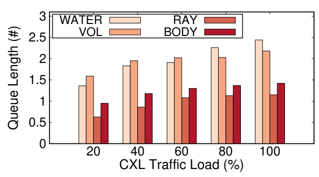
back-propagation approach can dissect the reduced CXL memory impact from LLC to L2/L1D. SB benefits from L1 data locality, with DWr path stalls decreasing by 90.6% on average from FlexBus+MC to SB. Besides, HWPF CXL-induced stalls on FlexBus+MC are correlated with DRd CXL-induced stalls on L1D and L2 (Figure 6). For instance, in BFS, there is a 353.5ns HWPF stall on the FlexBus+MC corresponding to 209.8ns and 179.46ns DRd stalls on L1D and L2, respectively. In contrast, for FREQ, we observe a 92.2 ns HWPF stall on FlexBus+MC with only a 13.4ns and 0.7ns stall on L1D and L2. PFEstimator can implicitly capture the effectiveness of L1D and L2 hardware prefetchers in mitigating the impact of CXL memory.

5.4 Case 3: Local v.s. CXL Access Interference

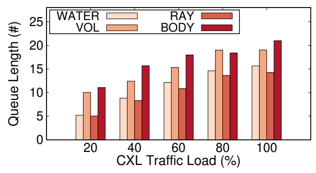
Although there is no memory channel contention between local mFlow and CXL mFlow, resource contention on other host components can cause severe interference. PathFinder can detect and analyze interference between local and CXL mFlows. We configure a case where a local mFlow and a CXL mFlow are located on the same core, and vary the CXL traffic load gradually from 20% to 100%. PathFinder first identifies potential interference between local and CXL mFlows by analyzing the uncore target request distribution reported by PFBuilder. PFAnalyzer and PFEstimator then dissect the interference. As shown in Figure 8-d, the queueing effect at the FlexBus and CHA stays stable. However, CXL-induced stall within a core is increased by 1.7×, 2.2×, 2.2×, 2.4×, and 2.4× on SB, L1D, LFB, L2, and core LLC, respectively (Figure 7-a/b/c/d/e). Even though FlexBus and CHA are not congested, PFEstimator captures the increased queueing within the core. PFAnalyzer further confirms the impact on core components by reporting rising queue lengths of LFB and L2 (Figure 8-b/c), particularly for the DRd path. PFAnalyzer also shows that increased CXL-induced stalls degrade L1D locality, which results in (a) heavy queueing at L2 and (b) long



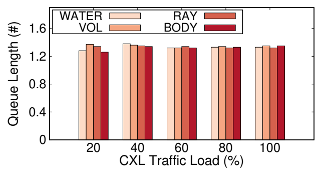
(a) L1D.



(b) LFB.



(c) L2 (DRd).



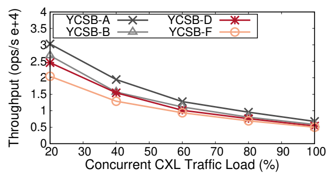
(d) FlexBus+MC (DRd).

Figure 8: We present the component queue length as varying the CXL traffic load. (a), (b), (c), (d) report the behavior for L1D, LFB, L2, and FlexBus+MC, respectively.

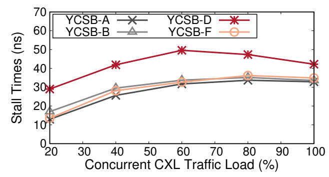
L2 request serve latency, making the core bottleneck shift from DRd on L1D to DRd on L2. A combination of path-level request distribution analysis, stall time back-propagation, and queue length estimation based on enqueue and dequeue rates enables dissecting interference between local and CXL mFlows and the resulting performance degradation of affected components.

5.5 Case 4: Concurrent CXL Access Contention

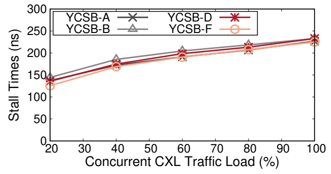
PathFinder can also analyze interference among CXL mFlows. PFEstimator leverages a back-propagation algorithm to compute CXL-induced stall time for each component, enabling it to capture how CXL mFlows interference pressure propagates backward from shared FlexBus to each core. In parallel, PFAnalyzer localizes the bottleneck and characterizes path states under neighboring CXL mFlows contention from core-issued requests and memory subsystem served requests. When CXL mFlow traffic increases from 20% to 100%, YCSB mFlow throughput is decreased by an average of 77.4% (Figure 9-a). All CXL mFlows originating from different cores aggregate at FlexBus+MC before reaching the CXL memory, and PFEstimator captures a 4.3× increase in FlexBus+MC latency (Figure 9-h), PFAnalyzer shows that the queueing degree of FlexBus+MC DRd and HWPF is increased by 4.6× and 1.2× (Figure 10-e/f), indicating that contention and blocking among CXL mFlows first manifest in the uncore FlexBus+MC. Figure 9-g shows that CHA latency rises by 1.6×, suggesting that (a) CHA cannot fully hide the interference across CXL mFlows; (b) the growing memory pressure from CXL memory propagates upward along the hierarchy and impacts core component performance. YCSB mFlow LLC experiences a 1.8× increase in CXL-induced stall time (Figure 9-f) and a 3.4× increase in queueing degree (Figure 10-d), suggesting CXL mFlows on LLC are not isolated and request interleaving occurs. Although CXL mFlows from different cores do not interleave at the individual core, the increased uncore interference still affects the core components. As shown in Figures 9-b/d/e, CXL-induced stall time on the SB, LFB, and L2 increases by 2.1×, 2.9×, and 1.8×, with 1.6× and 1.2× increased queueing degrees of LFB and L2 (Figure 10-b/c), indicating that uncore contention among CXL mFlows indeed indirectly affects private core components. When YCSB mFlow CXL-induced stall on the L1D rises by 1.7×(Figure 9-c), the L1D queueing degree



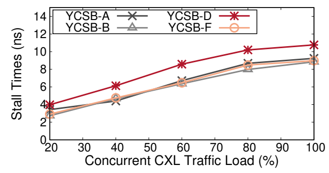
(a) Application Throughput.



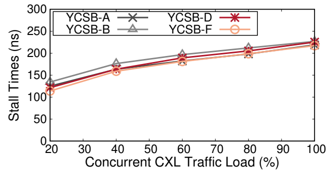
(b) SB.



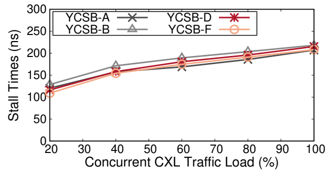
(c) L1D.



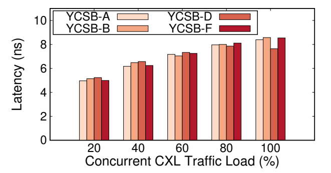
(d) LFB.



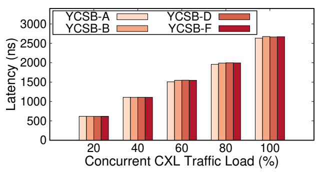
(e) L2.



(f) Core LLC.

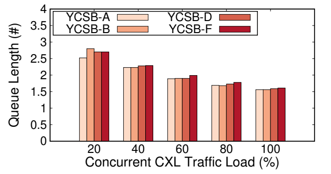


(g) CHA latency (DRd).

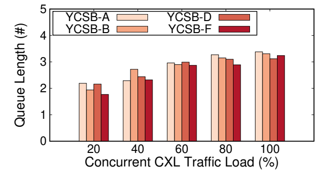


(h) FlexBus+MC latency (DRd).

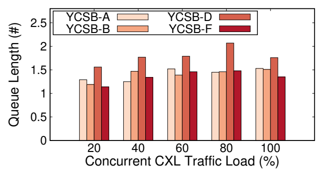
Figure 9: We present application performance and pipeline stall cycles while increasing the CXL mFlow load from 20% to 100%.



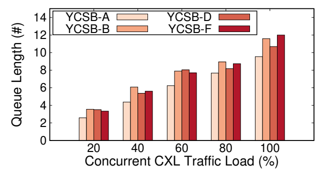
(a) L1D.



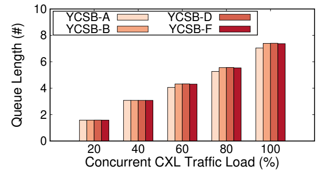
(b) LFB.



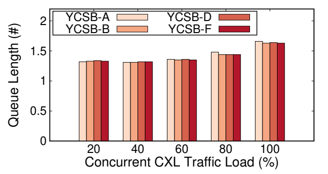
(c) L2.



(d) Core LLC.

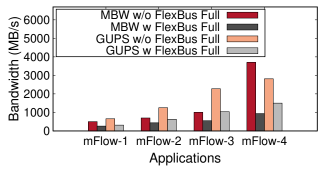


(e) FlexBus+MC (DRd).

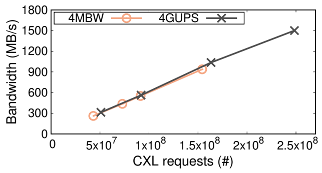


(f) FlexBus+MC (HWPF).

Figure 10: We present the queue length while increasing the CXL mFlow load from 20% to 100%. (a), (b), (c), and (d) present L1D, LFB, L2, and core-accessed LLC queue length. (e) and (f) present the FlexBus+MC queue length for DRd and HWPF, respectively.



(a) mFlows bandwidth.



(b) CXL BW and request frequency.

Figure 11: We examine the relationship between CXL bandwidth partition among concurrent CXL mFLows. (a) shows the results using 4 MBW programs with bandwidths of 500 MB/s, 700 MB/s, 1000 MB/s, and 3700 MB/s; using 4 GUPS programs with bandwidths of 650 MB/s, 1250 MB/s, 2200 MB/s, and 2800 MB/s. (b) shows the relationship between CXL memory request frequency and runtime bandwidth.

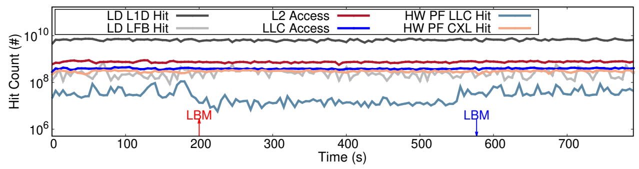
drops by 41.0% (Figure 10-a). PFAnalyzer accounts for the decreased core issue request rate and correctly identifies that L1D queueing decreases, while the mFlow bottleneck gradually shifts from DRd on L1D to HWPF on FlexBus+MC. In summary, PathFinder enables per-mFlow path-level analysis to identify interference, such as where the contention occurs, which request types dominate, and how each mFlow is indirectly affected in its core components.

5.6 Case 5: CXL Bandwidth Partition

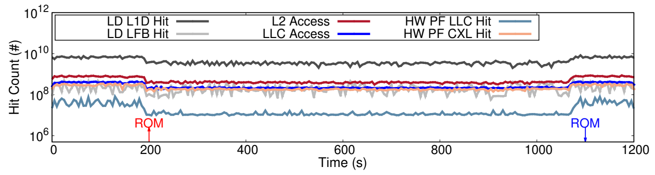
Understanding bandwidth allocation among concurrent CXL memory streams is a challenging task for application developers. PathFinder can assist in identifying the bandwidth among mFlows when the saturated CXL device bandwidth becomes a bottleneck for the host network. We evaluate this in 2 scenarios, i.e., 4 MBW instances contention and four GUPS instances contention, both cause FlexBus+MC to be saturated. Bandwidth competition reduces the bandwidth of each mFlow (Figure 11-a). But due to varying memory access patterns, the bandwidth degradation is not uniform across mFlows. For example, MBW-2 experiences a 37.7% drop, while MBW-4 is degraded by 74.7%. However, under severe FlexBus contention, PathFinder can estimate runtime bandwidth allocation based on each mFlow's demand on the uncore FlexBus. When the CXL memory access latency increases to 974.9ns and 753.6ns, severe FlexBus blocking occurs. PFAnalyzer detects the maximum queue length at FlexBus, indicating that each mFlow's performance is largely determined by FlexBus. Moreover, the system observes a Pearson correlation coefficient as high as 0.998 between each mFlow's CXL memory request frequency and its application-level reported bandwidth (Figure 11-b). Therefore, when PFAnalyzer identifies the culprit path at FlexBus+MC, the system can leverage PFBuilder's report of CXL memory request frequency to infer bandwidth allocation among concurrent CXL mFlows at runtime.

5.7 Case 6: Data Locality

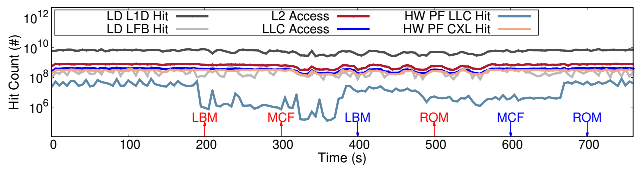
PathFinder can report the data locality changes across multiple snapshots using *PFMaterializer* (§4.6) to cluster the hit distribution of historical data, identify windows with stable memory access patterns, and report components with data locality changes when mFlow is disturbed. By walking through a sequence of snapshots, PathFinder examines how historical PMU counter statistics, captures the data locality trend, and further analyzes if co-located workloads contend the memory subsystem. For example, as shown in Figure 12, PathFinder shows that the LLC misses are decreased by 20.6% when co-locating with 519.*lbm_r* compared with the 554.*roms_r*



(a) 503.$bwaves\_r$ locality changes when launching 519.lbm_r.



(b) 503.$bwaves\_r$ locality changes when launching 554.$roms\_r$.



(c) 503.$bwaves\_r$ data locality under multiple applications.

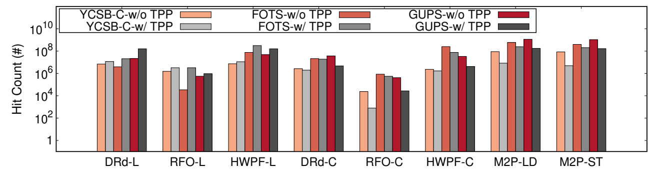
Figure 12: We monitor 503.$bwaves\_r$ application data locality changes when launching different applications. (a) Launch 519.lbm_r application accesses local memory. (b) Launch 554.$roms\_r$ application accesses CXL memory, further impacting 503.$bwaves\_r$ data locality. (c) A combination of 519.lbm_r, 505.mcf_r, and 554.$roms\_r$ applications accessing both local and CXL memory, introduce an additional interference effect to 503.$bwaves\_r$ data locality.

case, indicating that 503.*bwaves_r* is more execution-friendly when running with the 519.*lbm_r*.

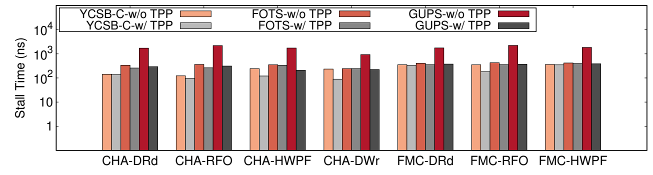
5.8 Case 7: Performance Optimization Using PathFinder

We first use PathFinder to understand how TPP [79] helps improve the application performance. When TPP is enabled, we observe that (a) YCSB-C, under the Zipf access pattern with a local/CXL memory ratio of 4:1, reduces its query latency by 2.5%; (b) GUPS with 24GB hot set, 72GB total working set, 1:1 read-write ratio, and 90% hot set access probability, under the same memory configuration, improves its throughput by 3.0×; and (c) 649.$fotonik3d\_s$ application reduces its execution time by 14.3% with a 2:1 local/CXL memory ratio.

As shown in Figure 13-a, PFBuilder traces collected from the core and M2PCIe PMU for YCSB-C, GUPS, and 649.$fotonik3d\_s$ demonstrates increased local memory access and decreased CXL memory access when TPP is enabled. For instance, GUPS shows a 7.4×/1.7×/3.3× increase in DRd/RFO/HWPF local memory hits from core PMU, while the corresponding hit counts on CXL memory decrease by 87.2%/93.4%/87.7%. For the M2PCIe PMU, closest to the CXL memory, load and store requests to the CXL memory are reduced by 84.6% and 84.4%, respectively. These results align with the TPP design, which migrates hot pages and shifts memory access towards local memory. We then use PFEstimator for further analysis. Figure 13-b shows that TPP reduces the average CHA access latency of DRd/RFO/HWPF/DWr requests and provides faster uncore serviceability. It observes the FlexBus+MC latency reduction, confirming that TPP constrains traffic along the CXL memory



(a) Hit event.



(b) Stall data path.

Figure 13: (a) shows hit event comparison of local and CXL memory for the YCSB-C, $649.fotonik3d\_s$, and GUPS applications with TPP disabled and enabled. (b) illustrates the stall data path comparison of CHA and FlexBus+MC with TPP disabled and enabled.

path, thus lowering overall uncore latency. Taking GUPS as an example, the latency of DRd/RFO/HWPF FlexBus+MC is reduced by 78.6%/83.5%/79.1%, resulting in an 82.9%/85.8%/88.0% corresponding uncore serving latency reduction. Last, we cross-validate with TPP by analyzing the application culprit path queueing degrees reported by PFAnalyser. When enabling TPP, YCSB-C culprit path (DRd on LFB) shows a 3.6% queueing degree decrease; the queueing of the GUPS culprit path (LD on FlexBus+MC) is dropped by 96.0%; and there is an 82.1% queueing reduction in the $649.fotonik3d\_s$ culprit path (DRd on L1D), which demonstrate that TPP's page promotion and demotion mechanism mitigates application bottlenecks.

We also use PathFinder to optimize the memory tiering mechanism. Take TPP+Colloid as an example [97]. Colloid aims to balance access latencies and guides TPP page migration at runtime using per-tier memory latency (i.e., CHA miss latency of DRd requests from different memory tiers). We explore a dynamic TPP+Colloid approach that PathFinder assists Colloid at runtime. It uses PFBuilder-reported CHA miss ratios of DRd/RFO/HWPF requests to select the most frequently accessed request type during the current execution phase, and then uses the corresponding local/CXL memory latency of the chosen type (obtained from PFEstimator) in place of Colloid's fixed DRd latency. This makes hot page migration better adapt to application memory access characteristics. Our evaluation shows that this approach can improve GUPS throughput by 1.1×.

5.9 Discussion

Limitation. Due to the lack of PMU counters specifically designed for RFO and HWPF requests within the core, PFBuilder and PFEstimator are limited. As shown in Table 7 produced by PFBuilder, we are unable to monitor RFO and DWr type requests at the L1D and LFB levels. The RFO counter at L2 indiscriminately includes both demand and prefetch RFO requests. PFEstimator faces similar issues when relying on core PMU data, the stall cycles at L1D, LFB, L2, and LLC are reported only for demand load requests and cannot be further broken down by access type. Future hardware with more advanced PMU capabilities would help resolve these issues.

System overheads. We measure the CPU cycles and memory footprint when enabling and disabling PathFinder. On average across all the workloads, it consumes 1.3% CPU cycles and 38MB of memory with a marginal impact on the application execution.

6 Related Work

Profiling Systems. People have developed many software utilities to identify code hotspots, analyze concurrency dependency, and break down stalled cycles [2, 8, 31, 37, 40, 42, 47, 48, 58, 59, 78, 96, 104]. Linux `perf` [40] is a widely used tool to instrument CPU performance counters, tracepoints, and report application execution statistics. Intel VTune [8] takes a top-down analysis strategy [104] and drills down the performance analysis using architectural counters. Some are also integrated into the language system, facilitating application development [2, 37, 48, 54, 61, 73, 76, 88–90, 92, 107]. PathFinder is built atop the Linux `perf` and PMUs.

Memory and Storage Disaggregation. People have explored disaggregated memory and storage extensively, given the rising networking bandwidth, fast remote storage protocol, and new cluster interconnect [49, 50, 56, 57, 62–64, 66–68, 71, 74, 75, 77, 79, 81–83, 91, 93–95, 99, 101, 106, 108]. For example, Ana Klimovic *et al.* characterize the performance of iSCSI-based disaggregated storage [62]. i10 [56] develops an efficient in-kernel TCP/IP remote storage based on dedicated end-to-end IO paths and delayed doorbell notifications. Pond [66] extends CXL memory into disaggregated memory pools and effectively uses the stranded and used memory. Caption [95] provides an in-depth comparison between CXL memory and NUMA emulated CXL memory, and proposes a CXL memory-aware page allocation policy that efficiently utilizes the CXL memory expander. Melody [68] extensively characterizes application access patterns on the CXL memory and proposes a runtime slowdown modeling framework to diagnose performance degradation code regions and components. We believe that PathFinder can help democratize the `CXL.mem protocol` and facilitate the deployment of CXL-based memory pooling.

Host Networking. Host interconnects have become a bottleneck under high-bandwidth networks. Researchers have developed benchmarking frameworks and diagnostic tools to analyze it [26–28, 52, 55, 70, 72]. For example, Saksham Agarwal et al. [28, 98] analyze the host congestion issues and build the host congestion control protocol. Hostping [70] monitors and diagnoses intra-host bottlenecks in RDMA networks. We focus on the `CXL.mem` protocol.

7 Conclusion

This paper presents a profiling and development utility for `CXL.mem`. It enables developers to understand and analyze the `CXL.mem` execution in an end-to-end manner. Our key idea is to view the server processor and its chipset as a multi-stage Clos network, equip each architectural module with a PMU-based telemetry engine, track different CXL.mem paths, and apply conventional traffic analysis techniques. We build PathFinder over Linux Perf and apply it to seven case studies. This work does not raise any ethical issues.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Stewart Grant, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199, CNS-2212192, CAREER-2339755, and Intel faculty awards.

References

[1] [n. d.]. Intel VTune Profiler. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html, year = 2025,. ([n. d.]).

[2] 2024. NVIDIA Nsight. https://developer.nvidia.com/tools-overview. (2024).

[3] 2025. AMD uProf. https://www.amd.com/en/developer/uprof.html. (2025).

[4] 2025. Hardware LLC prefetch feature on 4th Gen Intel Xeon Scalable Processor. https://www.intel.com/content/www/us/en/content-details/780991/hardware-llc-prefetch-feature-on-4th-gen-intel-xeon-scalable-processor-codename-sapphire-rapids.html. (2025).

[5] 2025. InfluxData: InfluxDB Time Series Data Platform. https://www.influxdata.com. (2025).

[6] 2025. Intel Compute Express Link (CXL) FPGA IP. https://www.intel.com/content/www/us/en/products/details/fpga/intellectual-property/interface-protocols/cxl-ip.html. (2025).

[7] 2025. Intel Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html. (2025).

[8] 2025. Intel VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. (2025).

[9] 2025. Intel Xeon Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html. (2025).

[10] 2025. IntelliProp's Omega Fabric. https://www.intelliprop.com/products-page. (2025).

[11] 2025. INTERNET CONTROL MESSAGE PROTOCOL. https://datatracker.ietf.org/doc/html/rfc792. (2025).

[12] 2025. "Memory directories" in Intel processors. https://sites.utexas.edu/jdm4372/2023/08/28/memory-directories-in-intel-processors/. (2025).

[13] 2025. Multichase: A Pointer Chaser Benchmark. https://www.amd.com/en/developer/uprof.html. (2025).

[14] 2025. Redis. https://redis.io/. (2025).

[15] 2025. Samsung CXL Memory Module - Box (CMM-B). https://semiconductor.samsung.com/news-events/tech-blog/cxl-memory-module-box-cmm-b/. (2025).

[16] 2025. SMART CXL Memory Modules. https://www.smartm.com/product/list/cxl-memory?utm_source=CXL&utm_medium=Website&utm_term=CXL-Website-TR&utm_content=CXL-Website-Link&utm_campaign=CXL-Website. (2025).

[17] 2025. SPEC CPU 2017. https://www.spec.org/cpu2017/. (2025).

[18] 2025. The CXL Specification. https://www.computeexpresslink.org/download-the-specification. (2025).

[19] 2025. The Falcon C5022. https://www.h3platform.com/product-detail/overview/35. (2025).

[20] 2025. The GigaIO FabreX Platform. https://gigaio.com/products/fabrex-system-overview/. (2025).

[21] 2025. The Leo CXL™ Memory Connectivity Platform. https://www.asteralabs.com/products/cxl-memory-platform/leo-cxl-memory-connectivity-platform/. (2025).

[22] 2025. The PCI Express (PCIe) Specification. https://pcisig.com/specifications/pciexpress/. (2025).

[23] 2025. Traceroute Wikipedia. https://en.wikipedia.org/wiki/Traceroute. (2025).

[24] 2025. UnifabriX MAX. https://www.unifabrix.com/technology. (2025).

[25] 2025. XConn Titan Evaluation Kit. https://www.xconn-tech.com/products. (2025).

[26] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher De Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, et al. 2022. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. 198–204.

[27] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. 2022. Understanding host interconnect congestion. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks (HotNets '22)*. Association for Computing Machinery, New York, NY, USA, 198–204. https://doi.org/10.1145/3563766.3564110

[28] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*. 275–287.

[29] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi, and Teh Ying Wah. 2015. Time-Series Clustering–a Decade Review. *Information systems* 53 (2015), 16–38.

[30] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (SIGCOMM'08)*. 63–74.

[31] Sotiris Apostolakis, Chris Kennelly, Xinliang David Li, and Parthasarathy Ranganathan. 2025. Necro-reaper: Pruning away Dead Memory Traffic in Warehouse-Scale Computers. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 689–703. https://doi.org/10.1145/3676641.3716007

[32] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing Router Buffers. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. 281–292.

[33] Scott Beamer, Krste Asanović, and David A. Patterson. 2015. The GAP Benchmark Suite. *ArXiv* abs/1508.03619 (2015). https://api.semanticscholar.org/CorpusID:11503794

[34] Daniel S Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D Hill, et al. 2023. Design tradeoffs in CXL-based memory pools for public cloud platforms. *IEEE Micro* 43, 2 (2023), 30–38.

[35] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 72–81.

[36] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT'19)*. 15–29.

[37] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. 2024. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*. 1–12.

[38] Charles Clos. 1953. A study of non-blocking switching networks. *Bell System Technical Journal* 32, 2 (1953), 406–424.

[39] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[40] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[41] Chen Ding and Yutao Zhong. 2003. Predicting Whole-Program Locality Through Reuse Distance Analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03)*. 245–257.

[42] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 727–741. https://doi.org/10.1145/3582016.3582031

[43] Pouya Esmaili-Dokht, Francesco Sgherzi, Valéria Soldera Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Monemi, Adrià Armejach, Estanislao Mercadal, Germán Llort, Petar Radojković, Miquel Moreto, Judit Giménez, Xavier Martorell, Eduard Ayguadé, Jesus Labarta, Emanuele Confalonieri, Rishabh Dubey, and Jason Adlard. 2024. A Mess of Memory System Benchmarking, Simulation and Application Profiling. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO'24)*. 136–152.

[44] Gettys, Jim and Nichols, Kathleen. 2012. Bufferbloat: dark buffers in the internet. *Commun. ACM* 55, 1 (2012), 57–65.

[45] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. 2023. Memory Pooling with CXL. *IEEE Micro* 43, 2 (2023), 48–57.

[46] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (ATC'22)*.

[47] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A call graph execution profiler. *SIGPLAN Not.* 17, 6 (jun 1982), 120–126. https://doi.org/10.1145/872726.806987

[48] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. 2023. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*. 916–929.

[49] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: a hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 417–433. https://doi.org/10.1145/3503222.3507762

[50] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. 2023. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*. 1012–1027.

[51] Daniel Hackenberg, Daniel Molka, and Wolfgang E Nagel. 2009. Comparing Cache Architectures and Coherency Protocols on X86-64 Multicore SMP Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on*

microarchitecture (MICRO'09). 413–422.

[52] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. 2022. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA, 861–877.

[53] James D Hamilton. 2020. *Time series analysis*.

[54] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. 2023. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*. 33–47.

[55] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. 2024. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*. 297–312.

[56] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 127–140.

[57] Sheng Jiang and Ming Liu. 2025. Building an Elastic Block Storage over EBOFs Using Shadow Views. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1137–1153.

[58] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[59] Yuyuan Kang and Ming Liu. 2025. Understanding and Profiling NVMe-over-TCP Using ntprof. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1117–1136.

[60] Ethan Katz-Bassett, Harsha V. Madhyastha, Vijay Kumar Adhikari, Colin Scott, Justine Sherry, Peter van Wesep, Thomas Anderson, and Arvind Krishnamurthy. 2010. Reverse traceroute. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*.

[61] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. 2010. SD3: A scalable approach to dynamic data-dependence profiling. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 535–546.

[62] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*.

[63] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 345–359.

[64] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G Shin. 2022. Hydra: Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 181–198.

[65] Philip Levis, Kun Lin, and Amy Tai. 2023. A Case Against CXL Memory Pooling. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets'23)*. 18–24.

[66] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: CXL-based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. 574–587.

[67] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. {ROLEX}: A Scalable {RDMA-oriented} Learned {Key-Value} Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114.

[68] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. 2025. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1203–1217. https://doi.org/10.1145/3676641.3715987

[69] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. 2023. Hostping: Diagnosing Intra-host Network Bottlenecks in RDMA Servers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*. 15–29.

[70] Kefei Liu, Zhuo Jiang, Jiao Zhang, Haoran Wei, Xiaolong Zhong, Lizhuang Tan, Tian Pan, and Tao Huang. 2023. Hostping: Diagnosing Intra-host Network Bottlenecks in RDMA Servers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA, 15–29.

[71] Ming Liu. 2020. *Building Distributed Systems Using Programmable Networks*. University of Washington.

[72] Ming Liu. 2023. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. 118–126.

[73] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*. 318–333.

[74] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. 2020. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. 305–323.

[75] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. 795–809.

[76] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 363–378.

[77] Liang Luo, Ming Liu, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2017. Motivating in-network aggregation for distributed deep neural network training. In *Workshop on Approximate Computing Across the Stack*.

[78] Zhihong Luo, Sam Son, Sylvia Ratnasamy, and Scott Shenker. 2024. Harvesting memory-bound CPU stall cycles in software with MSH. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation (OSDI'24)*. USENIX Association, USA, Article 4, 19 pages.

[79] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*.

[80] John D McCalpin. 1995. Stream benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what* 22, 7 (1995).

[81] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*. 106–122.

[82] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2023. eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. 461–477.

[83] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2024. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage* 20, 3, Article 16 (June 2024), 41 pages.

[84] Ashley O. Munch, Nevine Nassif, Carleton L. Molnar, Jason Crop, Rich Gammack, Chinmay P. Joshi, Goran Zelic, Kambiz Munshi, Min Huang, Charles R. Morganti, Sireesha Kandula, and Arijit Biswas. 2024. 2.3 Emerald Rapids: 5th-Generation Intel® Xeon® Scalable Processors. In *2024 IEEE International Solid-State Circuits Conference (ISSCC'24)*, Vol. 67. 40–42.

[85] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC'22)*, Vol. 65. 44–46.

[86] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*. 327–341.

[87] Gianna Paulin, Paul Scheffler, Thomas Benz, Matheus A. Cavalcante, Tim Fischer, Manuel Eggimann, Yichao Zhang, Nils Wistoff, Luca Bertaccini, Luca Colagrande, Gianmarco Ottavi, Frank K. Gürkaynak, Davide Rossi, and Luca Benini. 2024. Occamy: A 432-Core 28.1 DP-GFLOP/s/W 83% FPU Utilization Dual-Chiplet, Dual-HBM2E RISC-V-Based Accelerator for Stencil and Sparse Linear Algebra Computations with 8-to-64-bit Floating-Point Support in 12nm FinFET. In *IEEE Symposium on VLSI Technology and Circuits 2024, Honolulu, HI, USA, June 16-20, 2024*. 1–2.

[88] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. 2018. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 663–679.

[89] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. 2020. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*. 16–22.

[90] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. 2021. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. 772–787.

[91] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[92] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. 740–755.

[93] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. 1–16.

[94] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. 685–699.

[95] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*. 105–121.

[96] Yupeng Tang, Ping Zhou, Wenhui Zhang, Henry Hu, Qirui Yang, Hao Xiang, Tongping Liu, Jiaxin Shan, Ruoyun Huang, Cheng Zhao, Cheng Chen, Hui Zhang, Fei Liu, Shuai Zhang, Xiaoning Ding, and Jianjun Chen. 2024. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 818–833. https://doi.org/10.1145/3627703.3650061

[97] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key! *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3694715.3695968

[98] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. 2024. Understanding the Host Network. In *Proceedings of the ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24)*. Association for Computing Machinery, New York, NY, USA, 581–594. https://doi.org/10.1145/3651890.3672271

[99] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2020. Semeru: A {Memory-Disaggregated} managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 261–280.

[100] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad:{Non-Exclusive} Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*. 19–35.

[101] Xincheng Xie, Wentao Hou, Zerui Guo, and Ming Liu. 2025. Building Massive MIMO Baseband Processing on a Single-Node Supercomputer. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 1221–1242.

[102] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*. 130–149.

[103] Nofel Yaseen, John Sonchack, and Vincent Liu. 2018. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*. 402–416.

[104] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44.

[105] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. 2017. PARSEC3.0: A Multicore Benchmark Suite with Network Stacks and SPLASH-2X. *SIGARCH Comput. Archit. News* 44, 5 (Feb. 2017), 1–16. https://doi.org/10.1145/3053277.3053279

[106] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2022. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 37 (June 2022), 30 pages.

[107] Chenxingyu Zhao, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. 2025. White-Boxing RDMA with Packet-Granular Software Control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. 427–449.

[108] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*. 37–56.

Appendices are supporting material that has not been peer-reviewed.

Appendix A Appendix

A.1 Characterization Results on an Intel EMR Server

Figures 14, 15, 16 report the core PMU, CHA PMU, and uncore PMU counters when running characterization experiments on an Intel EMR machine.

A.2 Performance Counter Details

Tables 1, 2, 3, and 4 summarize the major performance counters used by PathFinder in the core PMU, CHA/LLC PMU, uncore PMU, and CXL device. Table 5 shows the counters used by *PFBuilder*.

A.3 PFEstimator Algorithm

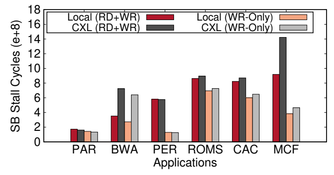
Algorithm 2 shows the details of PFEstimator.

A.4 Application Configuration

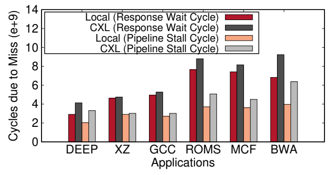
Table 6 summarizes our evaluated applications and their configurations.

A.5 More Evaluation Results

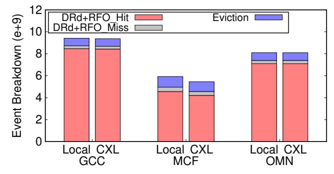
Table 7 shows path mapping for SPEC CPU2017 applications accessing CXL memory by PFBuilder.



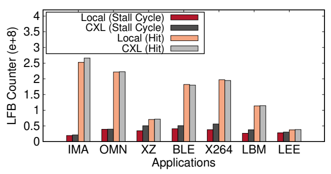
(a) Stall Buffer.



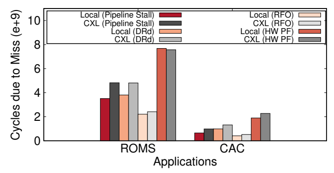
(b) L1D Execution.



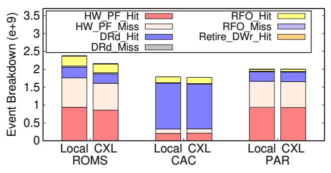
(c) L1D Operation.



(d) Line Fill Buffer.

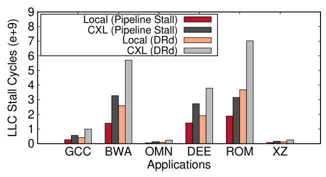


(e) L2 Execution.

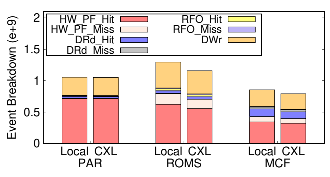


(f) L2 Operation.

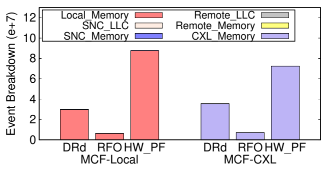
Figure 14: We compare core performance counters when running in the local and CXL memory cases on an Intel EMR machine. Our experimental configurations are similar to the ones used in Figure 2. (a) reports the core stall cycles of six applications when the store buffer becomes full under the read/write mixed and write-only cases. (b) and (c) present the L1D characteristics from the execution and operation perspectives, respectively. (d) shows the LFB counter statistics. (e) presents the core stall cycles and data responses under L2 misses, while (f) depicts the L2 operation breakdown.



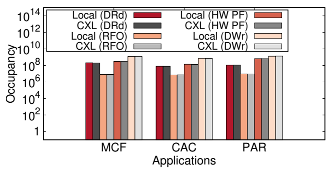
(a) Core LLC Execution.



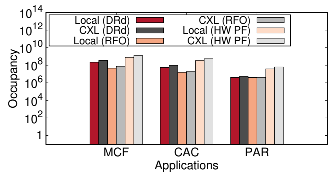
(b) Core LLC Hit/Miss Breakdown.



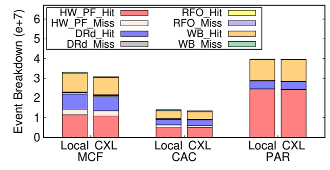
(c) Core LLC Miss Serve Target.



(d) CHA LLC Hit Occupancy.

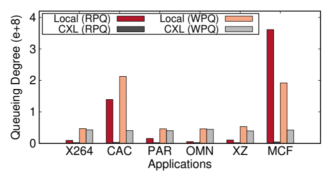


(e) CHA LLC Miss Occupancy.

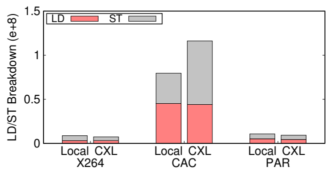


(f) CHA LLC Operation.

Figure 15: We compare CHA PMU when running in the local and CXL memory cases on an Intel EMR machine. Our experimental configurations are similar to the ones used in Figure 3. (a) reports the core LLC stall cycles. (b) presents the LLC hit/miss breakdown. (c) shows where the missed LLC requests are served. (d)/(e) report the occupancy of LLC hits and misses. (f) depicts the LLC operation breakdown. The y-axis of (d) and (e) is log-scale.



(a) Queueing Occupancy.



(b) Load/Store Breakdown.

Figure 16: We compare Uncore PMU of applications running in the local and CXL cases on an Intel EMR machine. Our experimental configurations are similar to the ones used in Figure 4. (a) reports the channel occupancy of RPQ and WPQ. (b) breaks down the load and store commands.

Architectural Component	Scope	Counter	Description
SB	per-core	resource_stalls.sb	Counts stall cycles caused by the store buffer being full where loads are still issued to avoid execution stall
	per-core	exe_activity.bound_on_stores	Counts cycles where the store buffer was full and no loads caused an execution stall
L1D	per-core	cycle_activity.cycles_l1d_miss	Counts cycles while L1D cache miss demand load is outstanding
	per-core	memory_activity.stalls_l1d_miss	Counts execution stall cycles while L1D cache miss demand load is outstanding
	per-core	l1d.replacement	Counts L1D data line eviction
	per-core	mem_load_retired.l1_hit	Counts retired load instructions hit in the L1D cache
	per-core	mem_load_retired.l1_miss	Counts retired load instructions missed in the L1D cache
LFB	per-core	mem_load_retired.fb_hit	Counts retired load instructions missed in L1 but hit LFB due to preceding miss to the same cache line
	per-core	l1d_pend_miss.fb_full	Counts number of cycles a demand request has waited due to LFB unavailability
L2	per-core	mem_load_retired.l2_hit	Counts retired load instructions with L2 cache hits as data sources
	per-core	mem_load_retired.l2_miss	Counts retired load instructions missed L2 cache as data sources
	per-core	mem_store_retired.l2_hit	Count retired store instructions hit L2 cache
	per-core	l2_rqsts.references	Counts all requests that were hit or true misses in L2 cache
	per-core	offcore_requests.all_requests	Counts memory transactions reached the super queue including requests initiated by the core
	per-core	l2_rqsts.all_demand_references	Counts demand requests to L2 cache
	per-core	l2_rqsts.all_demand_miss	Counts demand requests that miss L2 cache
	per-core	l2_rqsts.miss	Counts read requests of any type with true-miss in the L2 cache, excluding ongoing L2 misses
	per-core	offcore_requests.data_rd	Counts the demand and prefetch data reads
	per-core	l2_rqsts.all_demand_data_rd	Counts demand data read requests accessing the L2 cache, which may hit or miss L2 cache
	per-core	l2_rqsts.demand_data_rd_hit	Counts the number of demand data read requests initiated by load instructions that hit L2 cache
	per-core	offcore_requests.demand_data_rd	Counts the demand data read requests sent to uncore
	per-core	l2_rqsts.demand_data_rd_miss	Counts demand data read requests with true-miss in the L2 cache
	per-core	l2_rqsts.all_rfo	Counts the total number of RFO requests to L2 cache, including L1D RFO misses and L1D RFO prefetches
	per-core	l2_rqsts.rfo_hit	Counts the RFO requests that hit L2 cache
	per-core	l2_rqsts.rfo_miss	Counts the RFO requests that miss L2 cache
	per-core	l2_rqsts.swpf_hit	Counts software prefetch requests that hit the L2 cache
	per-core	l2_rqsts.swpf_miss	Counts software prefetch requests that miss the L2 cache
	per-core	memory_activity.stalls_l2_miss	Execution stalls while L2 cache miss demand cacheable load request is outstanding
	per-core	cycle_activity.cycles_l2_miss	Cycles while L2 cache miss demand load is outstanding
	per-core	ORO.data_rd	For every cycle, increments by the number of outstanding data read requests pending.
	per-core	ORO.cycles_with_data_rd	Counts cycles where at least 1 outstanding data read request is pending.
	per-core	ORO.demand_data_rd	For every cycle, increments by the number of outstanding demand data read requests pending
	per-core	ORO.cycles_with_demand_data_rd	Counts cycles where at least 1 outstanding demand data read request is pending.
	per-core	ORO.cycles_with_demand_rfo	Counts cycles where at least 1 outstanding demand RFO request is pending.
Latency	per-core	mem_trans_retired.load_latency	Loads latency starts by the actual cache access until the data is returned by the memory subsystem.
	per-core	mem_trans_retired.store_sample	Store latency starts when the demand write accesses the L1 datacache and lasts until the cacheline write is completed in the memory subsystem.

Table 1: The summary of key performance counters in the core PMU used by PathFinder. ORO=offcore_requests_outstanding.

Algorithm 2 The PFEstimator Algorithm.

```
1:  procedure PIPELINE_STALL_BREAKDOWN (CXL_DIMM_ID)
2:      id = CXL_DIMM_ID; AllLoads = 0;
3:      CXL_DIMM_Stall = get_CXLDIMM_stall_cycle(id);
4:      for j in all_FlexBus_RCs do                ▷ CXL DIMM → FlexBus RC
5:          FlexBusRC_Load[j] = load_agg_rc2dimm(j, i);
6:          AllLoads += FlexBusRC_Load[j];
7:      for j in all_FlexBus_RCs do
8:          weight = $\frac{FlexBusRC\_Load[j]}{AllLoads}$;
9:          FlexBusRC_Stall[j] = weight × CXL_DIMM_Stall;
10:     for i in all_FlexBus_RCs do                ▷ FlexBus RC → Host Uncore
11:         AllLoads = 0;
12:         FlexBusRC_Stall [i] += get_FlexBusRC_stall_cycle(i);
13:         for j in all_Host_Uncores do
14:             HostUncore_Load[j] = load_agg_uncore2rc(j, i);
15:             AllLoads += HostUncore_Load[j];
16:         for j in all_Host_Uncores do
17:             weight = $\frac{HostUncore\_Load[j]}{AllLoads}$;
18:             HostUncore_Stall[j] = weight × FlexBusRC_Stall[i];
19:     for i in all_Host_Uncores do               ▷ Host Uncore → CHA
20:         AllLoads = 0;
21:         HostUncore_Stall [i] += get_Uncore_stall_cycle(i, id);
22:         for j in all_CHAs do
23:             CHA_Load[j] = load_agg_cha2uncore(j, i);
24:             AllLoads += CHA_Load[j];
25:         for j in all_Host_Uncores do
26:             weight = $\frac{CHA\_Load[j]}{AllLoads}$;
27:             CHA_Stall[j] = HostUncore_Stall[i] × weight;
28:     CHA→LLC and in-core path segments are omitted;
```

Architectural Component	Scope	Counter	Description
Core LLC	per-core	cycle_activity.stalls_l3_miss	Counts execution stalls while L3 cache miss demand load is outstanding
	per-core	ORO.l3_miss_demand_data_rd	Counts the number of demand data read requests pending that are known to have missed the L3 cache
	per-core	mem_load_retired.l3_hit	Counts retired load instructions with at least one uop that hit in the L3 cache
	per-core	mem_load_retired.l3_miss	Counts retired load instructions with at least one uop that missed in the L3 cache
	per-core	mem_load_l3_hit_retired(4)	Counts retired load instructions whose data sources were HitM responses from shared L3, were L3 hit and cross-core snoop missed in on-pkg core cache, were L3 and cross-core snoop hits in on-pkg core cache and were hits in L3 without snoops required
	per-core	mem_load_l3_miss_retired(4)	Retired load instructions which data sources missed L3 but serviced from local DRAM, remote DRAM, forwarded from a remote cache or hitm in remote cache
	per-core	longest_lat_cache.miss	Counts core-originated cacheable requests that miss the L3 cache
	per-core	longest_lat_cache.reference	Counts core-originated cacheable requests to the L3 cache
	per-core	ocr.modified_write.any_response	Counts writebacks of modified cachelines and streaming stores that have any type of response
	per-core	ocr.demand_data_rd(9)	Counts offcore demand data reads in 9 scenarios: have any type of response; hit in the L3 or were snooped from another core's caches on the same socket; not supplied by the local socket's L1, L2, or L3 caches; supplied by DRAM attached to this socket, unless in Sub NUMA Cluster(SNC) Mode. In SNC Mode counts only those DRAM accesses that are controlled by the close SNC Cluster; hit in a distant L3 Cache or were snooped from a distant core's L1/L2 caches on this socket when the system is in SNC (sub-NUMA cluster) mode; supplied by DRAM on a distant memory controller of this socket when the system is in SNC (sub-NUMA cluster) mode; supplied by a cache on a remote socket where a snoop hit a line in another core's caches; supplied by DRAM attached to another socket; supplied by CXL DRAM
	per-core	ocr.rfo.any_response(9)	Counts offcore demand RFO in 9 scenarios same as DRd scenarios
	per-core	ocr.l1d_hw_pf(9)	Counts offcore L1D hardware prefetch in 9 scenarios same as DRd scenarios
	per-core	ocr.l2_hw_pf_drd(9)	Counts offcore L2 hardware prefetch DRd in 9 scenarios same as DRd scenarios
	per-core	ocr.l2_hw_pf_rfo(9)	Counts offcore L2 hardware prefetch RFO in 9 scenarios same as DRd scenarios
CHA LLC	per-socket	unc_cha_tor_inserts.ia(4)	Counts the number of entries successfully inserted into the TOR come from cores in 4 scenarios: total requests; requests hit LLC; requests miss LLC; requests miss LLC and target to CXL;
	per-socket	unc_cha_tor_inserts.ia_drd(9)	Count DRd from core in TOR in 8 scenarios: total insert; hit LLC; miss LLC; miss LLC and target DDR; miss LLC and target local; miss LLC and target local DDR; miss LLC and target remote; miss LLC and target remote DDR; miss LLC and target CXL;
	per-socket	unc_cha_tor_inserts.ia_drd_pref(9)	Count DRd prefetch from core in TOR in 9 scenarios same as DRd scenarios
	per-socket	unc_cha_tor_inserts.ia_rfo(6)	Count RFO from core in TOR in 6 scenarios: total insert; hit LLC; miss LLC; miss LLC and target local; miss LLC and target remote; miss LLC and target CXL;
	per-socket	unc_cha_tor_inserts.ia_rfo_pref(6)	Count RFO prefetch from core in TOR in 6 scenarios same as RFO scenarios
	per-socket	unc_cha_tor_inserts.ia_wb(5)	Count write back request from core in TOR in 5 scenarios: write back from E/F to E state; from E/F to I state; from M to E state; from M to I state; from S to I state;
	per-socket	unc_cha_tor_occupancy.ia(4)	For each cycle, this event accumulates the number of valid entries in the TOR that come from cores in 4 scenarios: total requests; requests hit LLC; requests miss LLC; requests miss LLC and target to CXL;
	per-socket	unc_cha_tor_occupancy.ia_drd(9)	For each cycle, this event accumulates the number of valid DRd entries in the TOR that come from cores in 9 scenarios: total insert; hit LLC; miss LLC; miss LLC and target DDR; miss LLC and target local; miss LLC and target local DDR; miss LLC and target remote; miss LLC and target remote DDR; miss LLC and target CXL;
	per-socket	unc_cha_tor_occupancy.ia_drd_pref(9)	For each cycle, this event accumulates the number of valid DRd prefetch entries in the TOR that come from cores in 9 scenarios same as DRd scenarios.
	per-socket	unc_cha_tor_occupancy.ia_rfo(6)	For each cycle, this event accumulates the number of valid RFO entries in the TOR that come from cores in 6 scenarios: total insert; hit LLC; miss LLC; miss LLC and target local; miss LLC and target remote; miss LLC and target CXL;
	per-socket	unc_cha_tor_occupancy.ia_rfo_pref(6)	For each cycle, this event accumulates the number of valid RFO prefetch entries in the TOR that come from cores in 6 scenarios same as RFO scenarios.
	per-socket	unc_cha_tor_occupancy.ia_wbmtoi	For each cycle, this event accumulates the number of valid write back M to I state entries in the TOR that come from cores.
	per-socket	unc_cha_tor_threshold1.ia(4)	Count the number of cycles subevent TOR not empty in 4 scenarios: total requests; requests hit LLC; requests miss LLC; requests miss LLC and target to CXL;
	per-socket	unc_cha_tor_threshold1.ia_drd(9)	Count the number of cycles subevent TOR not empty in 9 scenarios: total insert; hit LLC; miss LLC; miss LLC and target DDR; miss LLC and target local; miss LLC and target local DDR; miss LLC and target remote; miss LLC and target remote DDR; miss LLC and target CXL;
	per-socket	unc_cha_tor_threshold1.ia_drd_pref(9)	Count the number of cycles subevent TOR not empty in 9 scenarios same as DRd scenarios.
	per-socket	unc_cha_tor_threshold1.ia_rfo(6)	Count the number of cycles subevent TOR not empty in 6 scenarios: total insert; hit LLC; miss LLC; miss LLC and target local; miss LLC and target remote; miss LLC and target CXL;
	per-socket	unc_cha_tor_threshold1.ia_rfo_pref(6)	Count the number of cycles subevent TOR not empty in 6 scenarios same as RFO scenarios.

Table 2: The summary of key performance counters in the CHA/LLC used by PathFinder.

Architectural Component	Scope	Counter	Description
IMC	per-channel	unc_m_rpq_cycles_ne.pch0	Counts the number of cycles that the RPQ of pch0 is not empty
	per-channel	unc_m_rpq_cycles_ne.pch1	Counts the number of cycles that the RPQ of pch1 is not empty
	per-channel	unc_m_cas_count.all.pch0	Counts the total number of DRAM CAS commands issued on pch0 channel
	per-channel	unc_m_cas_count.all.pch1	Counts the total number of DRAM CAS commands issued on pch1 channel
	per-channel	unc_m_cas_count.rd.pch0	Counts the total number of DRAM Read CAS commands issued on pch0 channel
	per-channel	unc_m_cas_count.rd.pch1	Counts the total number of DRAM Read CAS commands issued on pch1 channel
	per-channel	unc_m_cas_count.wr.pch0	Counts the total number of DRAM Write CAS commands issued on pch0 channel
	per-channel	unc_m_cas_count.wr.pch1	Counts the total number of DRAM Write CAS commands issued on pch1 channel.
	per-channel	unc_m_rpq_inserts.pch0	Counts the number of allocations into the Read Pending Queue of PCH0
	per-channel	unc_m_rpq_inserts.pch1	Counts the number of allocations into the Read Pending Queue of PCH1
	per-channel	unc_m_rpq_occupancy_pch0	Accumulates the occupancies of the PCH0 Read Pending Queue each cycle
	per-channel	unc_m_rpq_occupancy_pch1	Accumulates the occupancies of the PCH1 Read Pending Queue each cycle
	per-channel	unc_m_wpq_cycles_ne.pch0	Counts the number of cycles that the WPQ of pch0 is not empty
	per-channel	unc_m_wpq_cycles_ne.pch1	Counts the number of cycles that the WPQ of pch1 is not empty
	per-channel	unc_m_wpq_inserts.pch0	Counts the number of allocations into the Write Pending Queue of PCH0
	per-channel	unc_m_wpq_inserts.pch1	Counts the number of allocations into the Write Pending Queue of PCH1
	per-channel	unc_m_wpq_occupancy_pch0	Accumulates the occupancies of the PCH0 Write Pending Queue each cycle
	per-channel	unc_m_wpq_occupancy_pch1	Accumulates the occupancies of the PCH1 Write Pending Queue each cycle
M2PCIe	per-socket	unc_m2p_rxc_cycles_ne.all	Counts the number of cycles when the M2PCIe ingress is not empty
	per-socket	unc_m2p_rxc_inserts.all	Counts the number of entries inserted into the M2PCIe ingress queue
	per-socket	unc_m2p_txc_inserts_ak	Counts the number of acknowledgements entries inserted into the M2PCIe egress queue
	per-socket	unc_m2p_txc_inserts_bl	Counts the number of block data entries inserted into the M2PCIe egress queue

Table 3: The summary of key performance counters in the Uncore used by PathFinder.

Architectural Component	Scope	Counter	Description
CXL	per-socket	unc_cxlcm_rxc_pack_buf_inserts	Number of Allocation to Mem Rxx Packing buffer
	per-socket	unc_cxlcm_rxc_pack_buf_inserts.mem_data	Number of Allocation to Mem Data Packing buffer
	per-socket	unc_cxlcm_rxc_pack_buf_full.mem_req	Number of cycles the Packing Buffer is Full
	per-socket	unc_cxlcm_rxc_pack_buf_full.mem_data	Number of cycles the Packing Buffer is Full
	per-socket	unc_cxlcm_rxc_pack_buf_ne.mem_req	Number of cycles of Not Empty for Mem Rxx Packing buffer
	per-socket	unc_cxlcm_rxc_pack_buf_ne.mem_data	Number of cycles of Not Empty for Mem Data Packing buffer
	per-socket	unc_cxlcm_txc_pack_buf_inserts.mem_req	Number of Allocation to Mem Rxx Packing buffer
	per-socket	unc_cxlcm_txc_pack_buf_inserts.mem_data	Number of Allocation to Mem Data Packing buffer

Table 4: The summary of key performance counters in the CXL device.

PMU	Counter	Data Path	Description
Core	ocr.demand_data_rd(9)	DRd	Counts DRds in 9 scenarios: (a) have any type of response; (b) hits in the L3 or were snooped from another core's caches on the same socket; (c) not supplied by the local socket's L1, L2, or L3 caches; (d) supplied by DRAM attached to this socket, unless in Sub NUMA Cluster (SNC) Mode count DRAM accesses that are controlled by the close SNC Cluster; (e) hit in a distant L3 Cache or were snooped from a distant core's L1/L2 caches on this socket when the system is in SNC mode; (f) supplied by DRAM on a distant memory controller of this socket when the system is in SNC mode; (g)supplied by a cache on a remote socket where a snoop hit a line in another core's caches; (h) supplied by DRAM attached to another socket; (i) supplied by CXL DRAM;

	ocr.rfo(9)
	ocr.l1d_hw_pf(9)
	ocr.l2_hw_pf_drd(9)
	ocr.l2_hw_pf_rfo(9)
		RFO
		HW PF (L1D) → DRd
		HW PF (L2) → DRd
		HW PF (L2) → RFO
			Counts RFOs in 9 scenarios same as DRd scenarios;
			Counts L1D hardware prefetch in 9 scenarios same as DRd scenarios;
			Counts L2 hardware prefetch DRd in 9 scenarios same as DRd scenarios;
			Counts L2 hardware prefetch RFO in 9 scenarios same as DRd scenarios

CHA	unc_cha_tor_inserts.ia_drd(9)	DRd	Count DRd from core in TOR in 8 scenarios: (a) total insert; (b) hit LLC; (c) miss LLC; (d) miss LLC and target DDR; (e) miss LLC and target local; (f) miss LLC and target local DDR; (g) miss LLC and target remote; (h) miss LLC and target remote DDR; (i) miss LLC and target CXL;
	unc_cha_tor_inserts.ia_rfo(6)	RFO	Count RFO from core in TOR in 5 scenarios: (a) total insert; (b) hit LLC; (c) miss LLC; (d) miss LLC and target local; (e) miss LLC and target remote; (f) miss LLC and target CXL;

	unc_cha_tor_inserts.ia_drd_pref(9)
	unc_cha_tor_inserts.ia_rfo_pref(6)
		HW/SW PF→DRd
		HW/SW PF→RFO
			Count DRd prefetch from core in TOR in 9 scenarios same as DRd scenarios;
			Count RFO prefetch from core in TOR in 6 scenarios same as RFO scenarios;

	unc_cha_tor_inserts.ia_wb(5)	DWr	Count write back request from core in TOR in 5 scenarios: (a) write back from E/F to E state; (b) from E/F to I state; (c) from M to E state; (d) from M to I state; (e) from S to I state;

Uncore	unc_m2p_txc_inserts_ak(1)
	unc_m2p_txc_inserts_bl(1)
		DWr
		DRd
			Counts the number of acknowledgments entries inserted into the M2PCIe egress queue;
			Counts the number of block data entries inserted into the M2PCIe egress queue;

Table 5: The key PMU counters used by *PFBuilder* to construct the path map for a mFlow. TOR=Table of Requests, which is a hardware queue in the CHA. The number in the parenthesis (2nd column) indicates the sub-event amount for different cases.

Benchmark Suite	Applications	Parallelism	Working Set (MB)
	500.perlbench_r (PER)	1-64	202.5
	502.gcc_r (GCC)	1-64	1366.9
	503.bwaves_r (BWA)	1-64	822.3
	505.mcf_r (MCF)	1-64	609.1
	507.cactuBSSN_r (CAC)	1-64	789.5
	508.namd_r (NAM)	1-64	162.5
	510.parest_r (PAR)	1-64	419.4
	511.povray_r (POV)	1-64	7.0
	519.lbm_r (LBM)	1-64	410.5
	520.omnetpp_r (OMN)	1-64	242.0
	521.wrf_r (WRF)	1-64	178.8
	523.xalancbmk_r (XAL)	1-64	481.0
	525.x264_r (X264)	1-64	156.0
	526.blender_r (BLE)	1-64	633.7
	527.cam4_r (CAM)	1-64	856.0
	531.deepsjeng_r (DEEP)	1-64	699.5
	538.imagick_r (IMA)	1-64	286.5
	541.leela_r (LEE)	1-64	24.7
	544.nab_r (NAB)	1-64	146.3
	548.exchange2_r (EXC)	1-64	2.5
	549.fotonik3d_r (FOT)	1-64	848.4
SPEC CPU2017	554.roms_r (ROMS)	1-64	841.6
	557.xz_r (XZ)	1-64	775.4
	600.perlbench_s(PERS)	1-64	202.5
	602.gcc_s(GCCS)	1-64	7620.2
	603.bwaves_s(BWAS)	1-64	11467.1
	605.mcf_s(MCFS)	1-64	3960.8
	607.cactuBSSN_s(CACS)	1-64	6724.0
	619.lbm_s(LBMS)	1-64	3224.5
	620.omnetpp_s(OMNS)	1-64	242.3
	621.wrf_s(WRFS)	1-64	177.8
	623.xalancbmk_s(XALS)	1-64	481.8
	625.x264_s(X264S)	1-64	156.0
	627.cam4_s(CAMS)	1-64	873.6
	628.pop2_s(POPS)	1-64	1434.3
	631.deepsjeng_s(DEES)	1-64	6879.5
	638.imagick_s(IMAS)	1-64	7007.8
	641.leela_s(LEES)	1-64	25.0
	644.nab_s(NABS)	1-64	561.3
	648.exchange2_s(EXCS)	1-64	2.5
	649.fotonik3d_s(FOTS)	1-64	9642.8
	654.roms_s(ROMSS)	1-64	10386.9
	657.xz_s(XZS)	1-64	15344.0
	blackscholes(BLACK)	1-64	612.0
	bodytrack(BODY)	1-64	32.9
	facesim(FACE)	1-64	304.3
	ferret(FER)	1-64	97.9
	fluidanimate(FLU)	1-64	519.5
	freqmine(FRE)	1-64	631.9
PARSEC	raytrace(RAY)	1-64	1282.7
	swaptions(SWA)	1-64	5.5
	vips(PVIPS)	1-64	37.5
	x264(PX264)	1-64	80.0
	canneal(CAN)	1-64	850.5
	dedup(DEDUP)	1-64	1443.0
	streamcluster(STREAM)	1-64	109.0
	barnes(BARN)	1-64	1584.0
	ocean_cp(OCEAN)	1-64	3546.5
	radiosity(RADIO)	1-64	1442.5
	raytrace(SRAY)	1-64	22.5
	volrend(VOL)	1-64	54.0
SPLASH2X	water_nsquared(WATN)	1-64	28.5
	water_spatial(WATS)	1-64	669.5
	fft(FFT)	1-64	12291.0
	lu_cb(LUCB)	1-64	502.0
	lu_ncb(LUNCB)	1-64	501.5
	radix(RADIX)	1-64	4097.5
	Breadth-First Search (BFS)	1-64	15778.0
	Single-Source Shortest Paths (SSSP)	1-64	36456.3
GAPBS	PageRank (PR)	1-64	12616.1
	Connected Components (CC)	1-64	12381.1
	Betweenness Centrality (BC)	1-64	13394.5
	Triangle Counting (TC)	1-64	21027.0

Table 6: Benchmarking applications and their configurations.

Hit Location	DRd			RFO			HW PF			DWr		
	FOTS	GCCS-s1	GCCS-s2	FOTS	GCCS-s1	GCCS-s2	FOTS	GCCS-s1	GCCS-s2	FOTS	GCCS-s1	GCCS-s2
SB										7.8E+08	8.5E+07	2.1E+09
L1D	4.7E+09	8.1E+08	4.5E+09									
LFB	3.1E+08	2.8E+08	1.3E+08									
L2	4.3E+07	1.7E+07	1.7E+08	4.4E+06	3.8E+05	2.1E+07	1.8E+08	1.4E+08	1.7E+08	2.3E+05	84345	1.1E+07
local LLC	5.4E+06	5.0E+06	1.3E+07	7.5E+04	7.7E+04	1.0E+06	2.5E+07	9.7E+06	5.9E+07			
snc LLC	9.4E+05	4.0E+05	8.1E+06	6.1E+03	1.0E+04	8.4E+05	1.8E+07	8.1E+06	6.3E+07	1.5E+08	6.5E+06	7.9E+07
remote LLC	13385	5290	6984	586	42	379	19031	10920	22646			
CXL Memory	2.5E+07	1.9E+06	8.1E+06	1.5E+06	935	4.1E+06	2.2E+08	1.8E+08	2.1E+07			

Table 7: We classify mFlows into DRd, RFO, HW PF and DWr data paths and separate path hit distribution on SB, L1D, LFB, and L2 components for $649.fotonik3d\_s$ and $602.gcc\_s$, where GCCS-s1 and GCCS-s2 are two selected snapshots from $602.gcc\_s$. For mFlows entering uncore region, we separate hit distribution on local LLC, SNC LLC, remote LLC, and CXL memory. L1D and LFB counters only provide hit count data at load path granularity.