Systematic CXL Memory Characterization and Performance Analysis at Scale

Jinshu Liu
Virginia Tech
Blacksburg, USA

Hamid Hadian
Virginia Tech
Blacksburg, USA

Yuyue Wang
Virginia Tech
Blacksburg, USA

Daniel S. Berger
Microsoft and University of Washington
Redmond, USA

Marie Nguyen
Samsung
San Jose, USA

Xun Jian
Virginia Tech
Blacksburg, USA

Sam H. Noh
Virginia Tech
Blacksburg, USA

Huaicheng Li
Virginia Tech
Blacksburg, USA

Abstract

Compute Express Link (CXL) has emerged as a pivotal interconnect for memory expansion. Despite its potential, the performance implications of CXL across devices, latency regimes, processors, and workloads remain underexplored. We present Melody*, a framework for systematic characterization and analysis of CXL memory performance.* Melody *builds on an extensive evaluation spanning 265 workloads, 4 real CXL devices, 7 latency levels, and 5 CPU platforms.* Melody *yields many insights: workload sensitivity to sub-μs CXL latencies (140-410ns), the first disclosure of CXL tail latencies, CPU tolerance to CXL latencies, a novel approach (*Spa*) for pinpointing CXL bottlenecks, and CPU prefetcher inefficiencies under CXL.*

CCS Concepts: **• Hardware → Emerging technologies**; **• Computer systems organization → Architectures**.

Keywords: Compute Express Link, CXL, Memory, Profiling

ACM Reference Format:
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 3 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3676641.3715987

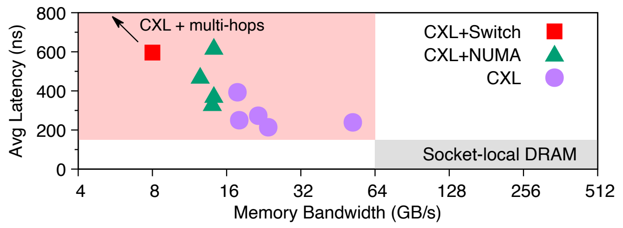

Figure 1. The spectrum of sub-μs CXL latency and bandwidth.

1 Introduction

Driven by the growing requirements of memory-intensive applications, the demand for increased memory capacity is rapidly rising [37]. The surge is further compounded by DRAM scaling challenges [40]. Emerging interconnects like Compute Express Link (CXL) hold the promise of both scale-up and scale-out memory expansion at the server/rack levels [34, 36, 44]. Various memory vendors have introduced CXL memory expanders [3, 4, 8, 15], some of which are being deployed in production systems, facilitating access to significantly larger amounts of DRAM than previously feasible.

Low memory access latency is key to system performance, but CXL memory expansion introduces higher latencies compared to traditional socket-local DRAM [27, 34, 41]. Figure 1 illustrates the substantial heterogeneity in CXL latency and bandwidth, as measured across 4 CXL devices within our platform (Table 1) and 2 more data points from public sources[1] [15, 17]. Furthermore, CXL devices can exhibit varying performance characteristics. The variability in latency and bandwidth arises from varying interconnection topologies and vendor optimizations [27, 41]. For instance, the latencies of locally-attached CXL range from ~200-400ns, slightly exceeding NUMA latency. Accessing CXL memory from a remote socket results in increased latency and diminished bandwidth (`CXL+NUMA`). The use of CXL switch(es) to extend connectivity will introduce additional latencies (`CXL+Switch`), even elevating latency to approximately 600ns.

[1] `CXL+Switch` data is from [15], and bandwidth is averaged for 1 CXL device.

The current CPU architecture and memory hierarchy are tailored for typical multi-socket systems, offering ~100ns latency and 100s of GB/s bandwidth. *However, the performance implications of CXL memory with sub-µs latencies remain largely unclear.* Currently, there is a significant gap in research that explores detailed CXL characteristics and their impact on memory-intensive workloads *at scale, in depth, and across the full spectrum of sub-µs latencies.*

In particular, how do CXL devices differ in detailed performance characteristics beyond average latency and bandwidth metrics? How (much) does CXL's long (and longer) latency affect CPU efficiency and workload performance? What are the underlying causes and how do we analyze it?

Simply treating CXL memory as slower DRAM is insufficient given the added complexity of the CXL protocol stack over PCIe and variability introduced by third-party memory controller optimizations [27]. While previous studies [28, 34, 38, 41–43] offer valuable insights into CXL performance impact on a small set of cloud/HPC workloads, they primarily focus on coarse-grained analysis and overlook several critical aspects: **(i)** CXL performance stability (*i.e.*, tail latencies); **(ii)** CPU tolerance to prolonged CXL latencies across various workloads, and the architectural implications of CXL; and **(iii)** the lack of systematic approach to dissect workload performance and CPU inefficiency under CXL.

To this end, we introduce **Melody**, a comprehensive framework for detailed CXL performance characterization: **(a)** Melody is the first to disclose, quantify, and analyze CXL tail latencies, providing insights through both microbenchmarks and real-world applications. **(b)** Melody evaluates 265 workloads across 4 CXL devices under 7 memory latency configurations (140-410ns) on 5 processor platforms. This extensive characterization provides insights into CPU and workload tolerance to CXL latencies. **(c)** Melody introduces a novel approach to diagnose CXL performance issues, utilizing only 9 CPU performance counters for lightweight, accurate and granular workload behavior analysis.

(I) The first analysis of CXL characteristics beyond average latency and bandwidth across 4 real CXL devices. Conventionally, memory performance has been characterized primarily by average latency and bandwidth. Most CXL studies to date have adhered to this traditional approach, implicitly assuming that CXL latencies would be as stable as those of local or NUMA memory [27, 28, 34, 36, 38, 41–43]. We discover some CXL devices exhibit significant µs-level tail latencies even under low device bandwidth utilization. We argue that tail latencies offer a crucial additional dimension for fully understanding CXL's characteristics, particularly given its PCIe-based, non-deterministic architecture and reliance on external memory controllers. Unlike tightly coupled integrated memory controllers (**iMC**) in CPUs, CXL setup may introduce greater latency variability.

(II) An extensive evaluation of CXL's performance implications across diverse workloads. Workloads have diverse memory demands and access patterns, resulting in varied behavior on CXL. A deep understanding of workload performance at scale is critical for informed deployment decisions [34, 43], as CXL latency can potentially lead to significant CPU stalls [33]. The limited scope of current studies restricts their broader applicability [41–43]. A large-scale study is needed to examine a more comprehensive set of CXL latencies and workloads. By providing extensive performance data on a wide range of real CXL devices, Melody facilitates cross-workload and cross-device analysis that were previously impossible. Furthermore, this large-scale evaluation allows us to systematically identify common performance patterns across workloads, which form the foundation of our root-cause analysis approach.

(III) A systematic approach for workload performance analysis under CXL. The root causes of CXL-induced performance slowdowns remain insufficiently understood. Existing research predominantly relies on heuristic-based, system-level metrics for correlation analysis, which often fail to accurately capture the underlying performance bottlenecks [26, 34, 39, 41, 45–47]. This is largely due to the opaque nature of CXL memory controllers and the semantic gap between architectural events and application behaviors. We introduce **Spa**, a novel analytical approach that repurposes CPU stall-related performance counters to pinpoint the sources of performance degradation under CXL and correlate them directly to workload behavior. Spa stands out due to its lightweight design (relying solely on 9 CPU counters), high accuracy (<5% inaccuracy for over 95% workloads), and robustness, validated across multiple CXL and NUMA configurations. The uniqueness of Spa lies in its ability to offer precise, low-overhead analysis of complex performance interactions under CXL, which is an important and powerful property that previous methods have been unable to achieve.

In summary, our core contributions are:

- **Melody**, the largest-scale CXL performance characterization to date, analyzing 265 workloads across 4 real CXL devices, 7 memory latency configurations, and 5 processor platforms. This analysis yields many insights into workload performance under sub-µs (CXL) memory latencies.
- The first disclosure and in-depth study of CXL tail latencies, assessing their impact on CPU efficiency and workload performance, and offering insights into CPU tolerance to extended CXL latencies.
- **Spa**: A novel CXL performance root-cause analysis approach based on CPU stall cycles using only 9 CPU counters. Spa enables detailed workload performance dissection under CXL, accurately diagnosing and quantifying various sources of CXL-induced performance degradations in the CPU at both the workload and execution-phase levels.
- The first comprehensive analysis of CPU hardware prefetcher inefficiencies under CXL's long latencies.
- Open-sourced Melody artifacts including tools and datasets at https://github.com/MoatLab/Melody.

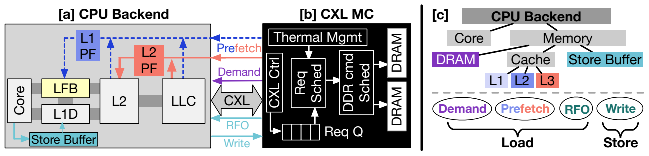

Figure 2. CPU and CXL. *CPU communicates with CXL controller via various types of load (demand, prefetch) and store (RFO, write) requests. "LFB" refers to Line Fill Buffer and "PF" denotes CPU prefetcher.*

2 Background

CXL for memory expansion. CXL [2] is an emerging memory fabric built atop PCIe, with memory expansion via the `CXL.mem` protocol being a key use case. At the microarchitecture level, the CXL protocol stack features a customized transaction layer responsible for queueing, processing, and ordering transactions while the link layer ensures reliable transmission through mechanisms like CRC and link-layer retries, maintaining data integrity across the Flex Bus link [2, 27]. CXL operates in full duplex, allowing simultaneous read and write operations, similar to cross-socket communications, and unlike the unidirectional DDR bus [27].

CXL memory controller (MC). Figure 2b illustrates CXL MC internals, which share many functional similarities with conventional DRAM controllers [31]. Memory requests to the CXL MC are encapsulated in a specific packet format, known as Flits [27], for transmission over CXL/PCIe. Upon arrival, the CXL controller ("CXL Ctrl") parses the request and places it in the request queue. The request scheduler then selects the next request to process based on the scheduling policy and other factors such as thermal management for low latency, high bandwidth, and reliability. Requests are then passed to the command scheduler, which issues appropriate low-level DDR commands to the DRAM chips.

Load/Store via CXL. Figure 2a shows components in the CPU backend [20] for DRAM/CXL request processing. If a request misses the cache (L1–L3/LLC), it will be forwarded to the CXL memory controller (**MC**). Caches are updated upon data returning from CXL MC. CPU's request processing flow remains the same for both local DRAM and CXL [27]. However, the use of different buses (DDR vs. CXL/PCIe) and MCs (iMC vs. third-party) affects CPU efficiency. Figure 2c classifies various request types. The CPU issues *two types of* `load` *requests*: *Demand* and *Prefetch*. Demand loads are memory reads that CPU requests from (CXL) MC only when it is needed for computation. Prefetch reads are predictive reads directed by prefetchers, *e.g.*, "L1PF" and "L2PF" in Figure 2a. Stores are first queued in the "store buffer." Each store request triggers a *Read-for-ownership (RFO)* for cache coherence from CXL/DRAM, followed by a *Write* upon cache eviction. Understanding the request types and processing flow are crucial for CXL performance analysis (§5).

In the rest of the paper, §3–§5 describe our CXL characterization and analysis, and we conclude in §6.

Table 1. Testbed. *"Local" and "Remote" denote inter- and cross-socket memory performance, respectively; #ch denotes the number of memory channels of each CPU/CXL.*

			Local		Remote		Specification
Server	#ch DDR	Size GB	Lat ns	BW GB/s	Lat ns	BW GB/s	#cores (Freq), L1D-L2-L3
SPR2S	8×DDR5	128	114	218	191	97	32 (2.1GHz), 48KB-2MB-60MB
EMR2S	8×DDR5	128	111	246	193	120	32 (2.1GHz), 48KB-2MB-160MB
EMR2S'	8×DDR5	1.5T	117	236	212	119	52 (2.3GHz), 48KB-2MB-260MB
SKX2S	6×DDR4	96	90	52	**140**	32	10 (2.2GHz), 32KB-1MB-13.8MB
SKX8S	6×DDR4	48	81	109	**410**	7	28 (2.5GHz), 32KB-1MB-38.5MB
CXL	#ch DDR	Size GB	Lat ns	BW GB/s	Lat ns	BW GB/s	Type, CXL-Spec, Server
CXL-A	2×DDR4	128	**214**	24	375	14	ASIC, CXL 1.1×8, SPR2S/EMR2S
CXL-B	1×DDR5	128	**271**	22	473	13	ASIC, CXL 1.1×8, SPR2S/EMR2S
CXL-C	2×DDR4	16	**394**	18	621	14	FPGA, CXL 1.1×8, SPR2S/EMR2S
CXL-D	2×DDR5	756	**239**	52	333	14	ASIC, CXL 1.1×16, EMR2S'

3 CXL Device Characterization

3.1 Testbed

Servers. Table 1 shows the details of our testbed. We use five servers with Intel processors: one SPR (SPR2S), two EMR (EMR2S/EMR2S'), and two SKX (SKX2S/SKX8S). The servers have 2 or 8 sockets (2S/8S) each with 6 or 8 memory channels. We use NUMA to simulate 3 additional CXL latency configurations of 140ns and 190ns on SKX2S (190ns achieved via lowering uncore frequency, not shown in the table), and 410ns on SKX8S [32, 34]. These setups provide a total of 7 latency configurations (bold texts in "Lat" column, Table 1).

CXL devices. We use 4 CXL memory expanders from different vendors (**CXL-A**, **CXL-B**, **CXL-C**, **CXL-D**). CXL devices are hosted on SPR/EMR servers: CXL-A, CXL-B, and CXL-C on SPR2S/EMR2S, and CXL-D on EMR2S' (remote host). Our CXL devices' **average** latency and bandwidth are 214-394ns and 18-52GB/s (Table 1, column "Local"), respectively, measured by Intel Memory Latency Checker (MLC, "`-latency_matrix` and `-bandwidth_matrix`") [6]. Since CXL links are full-duplex, the maximum achievable bandwidth for each CXL device is higher under read/write workloads. Our CXL devices (A-D) support peak bandwidths of 32GB/s, 26GB/s, 21GB/s, and 59GB/s, respectively (see Figure 5). All our CXL devices operate in CXL 1.1 mode as type-3 memory expanders (`CXL.io+CXL.mem`)[2]. CXL-C is FPGA-based (lowest performance) while the rest are ASICs. CXL-D utilizes 16× PCIe 5 lanes and 2 memory channels, providing the highest CXL bandwidth of 52GB/s. In contrast, the other devices use 8× lanes and 1 memory channel, resulting in nearly half the bandwidth (18-24GB/s) of CXL-D. CXL-A and CXL-C use DDR4 memory, while CXL-B and CXL-D use DDR5. CXL-A exhibits the lowest latency at 214ns, while DDR5-based CXL-B and CXL-D have higher latencies of 271ns and 239ns, respectively. The performance differences are due to vendor-specific optimizations (*e.g.*, scheduling policies, row buffer and thermal

[2]Our devices are CXL 2.0 capable but the CPUs only support CXL 1.1 mode.

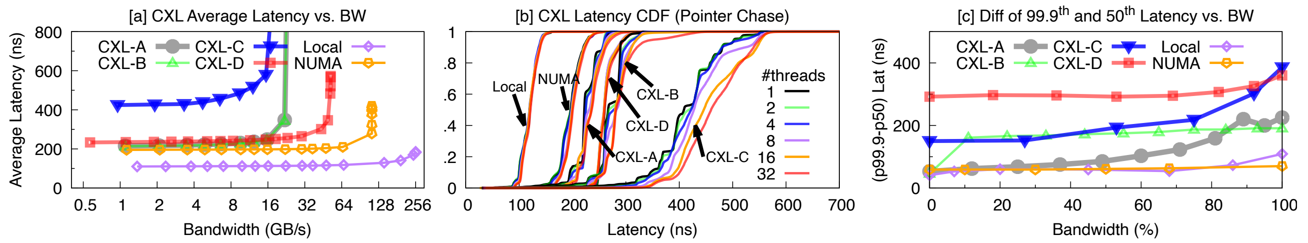

Figure 3. CXL Latency CDF. *Not all CXL are created equal. Unlike local/NUMA memory, CXL shows unstable and high tail latencies.*

management) [31, 41]. Accessing CXL from a remote socket (`Remote`) increases latency and reduces bandwidth. However, the latency increase via one NUMA hop varies significantly by device, *i.e.*, increasing by 161ns, 202ns, 227ns, and 94ns, for CXL A–D, respectively. Later, we show `CXL+NUMA` leads to unexpected slowdowns for some workloads (§4) which requires careful management.

Workloads. We use a diverse set of representative workloads for the characterization, covering cloud workloads (in-memory caching and databases such as Redis [13] and VoltDB [21], CloudSuite [1], and Phoronix [12]), graph processing (GAPBS [22], PBBS [19]), data analytics (Spark [30]), ML/AI (GPT-2 [5], MLPerf [14], Llama [9]), SPEC CPU 2017 [18], and PARSEC [24]. Some of the workloads are latency-sensitive (*e.g.*, many cloud workloads), approximately one quarter are bandwidth-sensitive (*e.g.*, HPC), while others are a mix of both. We consider a large-scale study essential to uncover key findings and insights (discussed later) that would not have been achievable with a small-scale study.

Performance metric. We run workloads using local, NUMA or CXL memory to focus on analyzing the "worst-case" CXL setup, excluding more complex configurations like tiering or interleaving. Local DRAM performance serves as the baseline for calculating CXL slowdowns (S), represented as the performance ratio between CXL and local DRAM that reflects the combined impact from increased latency and reduced bandwidth. *i.e.*, $S = (\frac{P_{\text{DRAM}}}{P_{\text{CXL}}} - 1) * 100\%$, where P represents workload performance under either DRAM or CXL.

3.2 CXL (Tail) Latencies and Bandwidth

As prior works have already demystified CXL average latency and bandwidth [41, 42], our focus is on CXL latency stability and its relationship with bandwidth.

CXL loaded latencies. To understand the CXL device-level performance characteristics, we first use Intel MLC [6] to measure average loaded latencies when the foreground latency measurement thread is under contention with 31 traffic generating threads, each of which injects delays of 0-20K cycles to simulate computation in between two adjacent memory accesses. Loaded latency refers to memory access latency under high utilization, in contrast to idle latency, which occurs when the system experiences minimal load. Figure 3a shows the latency-bandwidth curves. The average memory latency remains relatively stable when bandwidth utilization is low (*e.g.*, <50%) for all the setups. However, as utilization approaches the saturation point, latency begins to rise, accelerating significantly due to queuing delays once the bandwidth limit is exceeded (the vertical part at the right end of each line). While this is expected behavior, the rate of latency increase varies. In particular, **some CXL devices struggle to maintain stable latencies under load.** Before bandwidth saturation, CXL devices show significant latency increases as load rises, with behavior varying across devices compared to local/NUMA. For example, CXL-D manages latency effectively near saturation, like local/NUMA. In contrast, CXL-A and CXL-B see latency spikes from ∼350ns to ∼1.2µs, and CXL-C reaches 3µs when injected delay drops from 700 to 500 cycles. Notably, average latency increases by at least 60ns at 50-86% bandwidth utilization for CXL, while local/NUMA maintain stability at 90-95%, indicating CXL's higher sensitivity to bandwidth pressure.

CXL tail latencies. To investigate latency variability across CXL devices, we measure cacheline-level latencies using a custom microbenchmark, **MIO**, as existing tools lack request-level latency reporting. MIO, validated against Intel MLC for accuracy, measures the average latency of every N pointer-chase operations (configurable to reduce `rdtsc` overhead) on a working set larger than LLC. Latency logs are stored in a buffer in an idle NUMA node to minimize interference. Figure 3b displays the latency distributions for all 4 CXL devices and local/NUMA memory under 1-32 co-located pointer-chase threads. Pointer-chase is purely latency-sensitive and none of the CXL devices exceed 50% bandwidth. We disable CPU L1/L2 prefetchers to measure device-level latencies.

We find that CXL-B and CXL-C suffer from significantly high tail latencies, while local and NUMA memory show stable performance, with p99.9 and p50 latency differences of only 45ns and 61ns, respectively. In contrast, CXL latency stability varies significantly across vendors. The small variations in local and NUMA latencies likely stem from DRAM chip-level factors (*e.g.*, row buffer misses), as discussed in prior DRAM studies [25, 26, 29, 39, 46]. CXL devices show much larger latency variations. For instance, CXL-D demonstrates the best latency stability among all the CXL devices, with a p99.9-to-p50 difference of 75ns - just 30ns and 14ns more than local and NUMA. In contrast, CXL-B and CXL-C

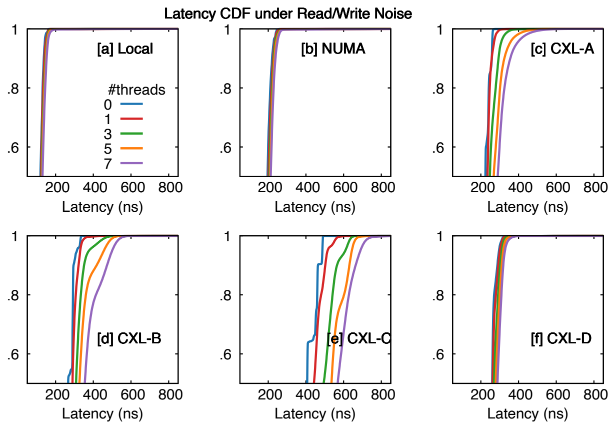

Figure 4. CXL Latency under Noise. *CXL latency is unstable under read/write traffic; "#threads" denotes the # of background threads generating traffic without saturating the device bandwidth; Three out of four CXL devices exhibit unstable and high tail latencies.*

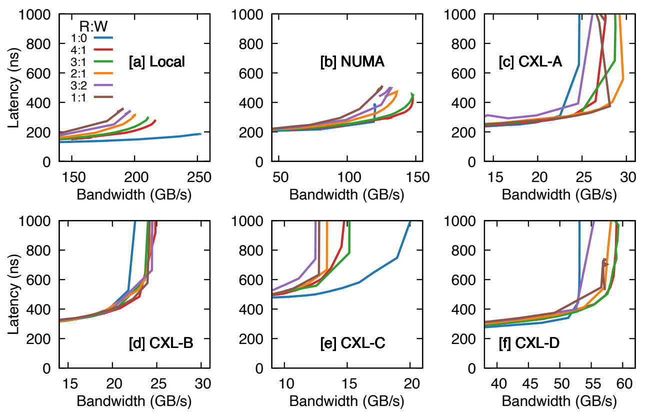

Figure 5. Latency-BW curves under various read/write ratios. *The performance patterns under various read/write ratios vary significantly across Local-DRAM, NUMA, and CXL.*

show differences of up to ∼160ns, a 50% increase over the median latency. At higher percentiles such as p99.99 and p99.999, CXL-A and CXL-D exceed 700ns, while CXL-B and CXL-C reach 1µs.

Figure 3c shows tail latency results at different levels of device bandwidth utilization. Bandwidth pressure is applied using multiple background read threads while a foreground thread performs point-chase. The Y-axis represents the difference between p99.9 and p50 latencies to highlight tail latency variation between CXL and local/NUMA memory. Under local/NUMA, p99.9 and p50 latencies remain stable, even at 90% bandwidth utilization. In contrast, CXL devices, specifically CXL-A and CXL-D, exhibit an increasing gap between p99.9 and p50 latencies, beginning at 30% and 70% utilization, respectively. However, stable latencies for CXL-A and CXL-D are seen in Figure 3b.

We further co-locate the foreground pointer-chase thread with multiple bandwidth-intensive read/write traffic generators using AVX instructions. Despite the device bandwidth not being fully saturated, we observe even worse tail latencies. The results in Figure 4 reveal a similar trend: while local and NUMA latencies remain stable, three out of four CXL devices exhibit significant latency variations, particularly at high percentiles (*i.e.*, the tails). For instance, CXL-A and CXL-B show a worsening trend in high-percentile latencies as the number of background threads increases. Furthermore, by reducing the number of server DIMMs per-socket from 8 to 2 to match that of CXL devices for a fair CXL/NUMA comparison as in [42], we consistently observe CXL tail latencies while not in local/NUMA (results not shown).

CXL latency vs. bandwidth under various read/write ratios. To thoroughly understand the latency-bandwidth relationship, we measure memory latency and bandwidth using 31 threads while varying the read/write ratios (1:0, 4:1, 3:1, 2:1, 3:2, and 1:1). Each thread generates read or write memory traffic at different intensities by injecting delays ranging from 0 to 40K cycles. Figure 5 presents the detailed latency-bandwidth results. We make the following key observations: **(1)** As expected, local DRAM achieves the highest bandwidth under a read-only workload, whereas NUMA and all CXL devices (except CXL-C) achieve minimal bandwidth in read-only scenarios. This is because NUMA and CXL links are bidirectional, allowing them to sustain higher bandwidth under mixed read/write workloads [41, 42]. **(2)** The FPGA-based CXL-C device cannot exploit CXL's bidirectional data transfer, achieving its highest bandwidth under a read-only workload, while increasing write ratios degrade its performance. We speculate that this is due to the unoptimized CXL IP on the FPGA being unable to utilize both CXL data transmission links, leading to behavior similar to local DRAM. **(3)** Unlike local DRAM and NUMA, which exhibit consistent performance trends as read/write ratios increase, CXL devices demonstrate significant variability. The peak bandwidth differs across CXL devices, occurring at 3:1/4:1 for CXL-D and 2:1 for CXL-A. Additionally, CXL devices exhibit more bandwidth fluctuations across different read/write ratios, underscoring the heterogeneity of CXL memory. These variations highlight the importance of careful tuning and workload-aware optimizations when deploying CXL-based memory solutions, especially for write-intensive workloads. Later in §4, we will show how the device-level heterogeneity manifest at the workload level.

Impact of CPU prefetchers on (tail) latency. Figure 6 demonstrates how enabling CPU prefetchers affects latency, revealing that prefetching does not fully mitigate CXL-induced tail latencies. Similarly, the results reveal significant performance disparities among local DRAM, NUMA, and CXL devices. Local DRAM and NUMA exhibit the lowest and most stable latencies, showing minimal variance even as thread counts increase, indicated by the sharp CDF curves. NUMA memory shows slightly higher latency than local DRAM due to the added cross-socket access overhead, but it remains relatively stable (*i.e.*, p99 < 50ns). In contrast, CXL devices

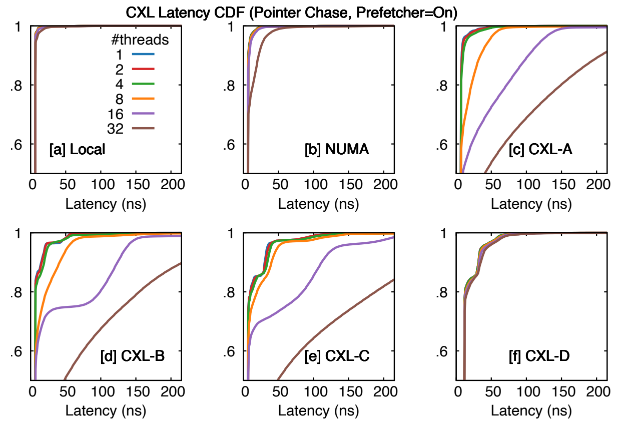

Figure 6. CXL latency under prefetchers. *Prefetching is insufficient to hide CXL-induced latencies.*

show considerably higher and more variable latencies, with CXL-A, CXL-B, and CXL-C displaying particularly long tail latencies. As the number of threads increases, CXL memory latencies further diverge, highlighting increased queuing effects and contention, whereas local and NUMA memory remain stable. Moreover, despite enabling hardware prefetching, CXL devices continue to experience significant tail latencies, suggesting that prefetchers are ineffective in mitigating CXL-induced delays for latency-sensitive workloads. These findings underscore the need for improved memory management strategies to address CXL's high latency variability and tail latency issues, particularly for latency-sensitive applications. In §5, we will formally quantify and reason about prefetcher inefficiencies under CXL using a novel root-cause analysis approach.

CXL tail latencies in real workloads. We sample memory latencies every 1ms across 112 workloads. Similarly, we find unpredictable CXL latencies for many workloads. In Figure 7a&b, workload `508.namd_r` bandwidth is mostly <500MB/s with a few spikes up to 3.4GB/s, yet, CXL-C still exhibits spiky latencies up to 1µs, indicating CXL MC's inability to maintain stable latencies. Note only 2s of data is shown for brevity. Figure 7c shows the tail latencies of the YCSB-C workload on Redis. The results reveal high tail latencies on CXL-C, while local/NUMA and CXL-B exhibit much lower tail latencies. YCSB-C, being a memory latency-sensitive read-only workload, suffers from elevated tail latencies on CXL-C because device-level latencies propagate to the application level, adversely impacting workload performance.

Reasoning. Understanding the source of CXL tail latencies is challenging due to the black-box nature of CXL devices. We first identify possible causes based on publicly available CXL and JEDEC specifications, and then design experiments to validate or rule out certain factors. CXL's unpredictable latencies can be attributed to several key issues: **(1) CXL's non-deterministic nature:** The CXL architecture inherently exhibits latency variability due to its customized transaction and link layers. For instance, according to the CXL specification, CXL link congestion (*e.g.*, due to flow control back-pressure) can lead to performance fluctuations, even under consistent, light loads [2]. While the transaction and link layers only take single-digit ns to process [27], the congestion effect might accumulate and lead to high queueing delays. Furthermore, mechanisms like transaction layer queuing delays add to this unpredictability [2]. This could help explain the latency unpredictability observed in CXL-C. We find that CXL-C was unable to fully utilize the bidirectional CXL link, unlike the other CXL devices by measuring maximum bandwidth under different read/write ratios (Figure 5). **(2) Unpredictability induced by CXL MCs (*e.g.*, thermal management):** According to the CXL specification and JEDEC CXL standard [7], CXL MCs can experience temporary performance issues due to power/thermal constraints. or DRAM refresh operations. These MC-level factors, combined with CXL protocol-level non-determinism, can amplify latency variability and result in severe performance degradation. We stress tested the CXL devices to investigate the impact of thermal and power constraints on latency but we did not observe a significant increase in tail latencies when repeating experiments from Figure 3c under 70°C. We did not further stress the temperature to avoid damaging the devices. Nonetheless, we believe thermal throttling could still be a potential cause of tail latencies as future PCIe 6.0 devices are expected to have higher power consumption [10, 11]. **(3) Suboptimal optimizations of existing CXL MCs:** Our results in Figure 3 show CXL MCs may not yet match the maturity and optimization level of iMCs, which have been fine-tuned over years for maximizing bandwidth and minimizing latency. High CXL tail latencies could be from suboptimal optimizations in the CXL MC (§2), for example, memory request scheduling could lead to temporary queuing delays. Currently, no tools exist to pinpoint tail latencies. A future approach could involve a white-box analysis, breaking down the latency of each memory request across components such as the CXL link, MC, and DRAM chips. This would require the CXL MC to expose detailed performance counters, potentially through the upcoming CXL Performance Monitoring Unit (CPMU) introduced in CXL 3.0 [2]. As an initial step, we demonstrate and quantify the impact of CXL tail latency to raise awareness within

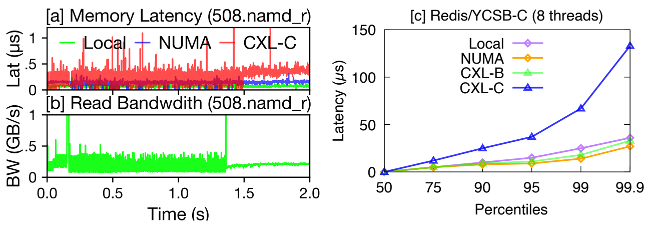

Figure 7. Latency of real workloads. *(a) CXL tail latencies can be observed even under low memory bandwidth utilization; (b) CXL device-level tail latency can propogate to application level (e.g., Redis).*

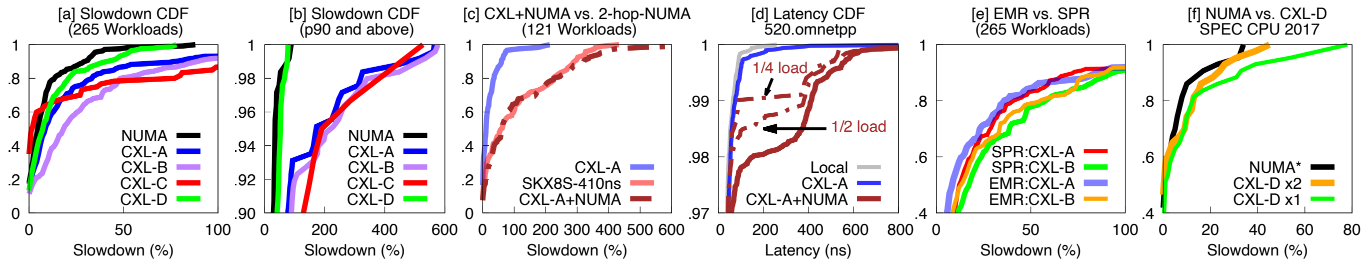

Figure 8. CXL workload slowdowns. *(a&b) CDFs of 265 workloads on 4 CXL devices; (c&d) shows tail latency impact under* `CXL+NUMA`*; (e) SPR vs. EMR performance comparison under CXL-A and CXL-B; (f) comparison of* `NUMA`*, CXL-D, and hardware interleaving of two CXL-Ds.*

the systems community. In sum, the key takeaways of CXL device characterization are:

Finding #1: (a) Not all CXL devices are created equal; each comes with unique performance characteristics not only in terms of average latency and bandwidth, but also in latency stability and the latency-bandwidth relationships. **(b)** More importantly, CXL devices exhibit unstable and higher tail latency compared to local/NUMA memory. Average latency and bandwidth do not fully capture the performance implications of CXL devices. CXL MCs fail to mitigate tail latencies under light load, and high memory utilizations will further exacerbate CXL tail latencies, while local memory and NUMA maintain stable latencies. **(c)** Concurrent reads/writes affect CXL latency differently, with tail latency worsening in mixed workloads. **(d)** While CPU hardware prefetchers can improve average memory access latencies, they fail to eliminate tail latencies. CXL tail latencies negatively impact application performance. **(e)** FPGA-based CXL device fails to fully leverage CXL's bidirectional data transfer capabilities, resulting in performance characteristics that differ significantly from ASIC-based devices under mixed read/write workloads. The peak bandwidth of CXL devices varies across different read/write ratios.

Implication #1: From software/hardware design perspectives, there is a need to address CXL tail latencies. **(a)** Future CPUs need to be improved to tolerate CXL's unpredictable latencies as they could stall the CPU longer, possibly leading to cascading performance degradations under dependent data access. **(b)** CXL MC designs need to prioritize latency predictability alongside average bandwidth and latency. **(c)** System designers need to account for tail latency impact when developing and deploying software on CXL.

Recommendation #1: Tail latency should be used as a key metric for evaluating CXL devices, as predictable latency is crucial for quality of service (QoS) in the cloud.

4 Workload Characterization

CXL slowdowns across devices. Figure 8a presents the CDFs of CXL slowdown across all workloads for 4 CXL devices and NUMA on EMR.[3] Overall, *CXL devices exhibit slower performance than NUMA due to higher latencies and lower bandwidths; however, the highest-performance CXL device (CXL-D) performs almost as well as with NUMA.* As CXL latency increases (D→A→B→C), slowdowns consistently worsen across all workloads. Under NUMA (193ns, 120GB/s), 98% of the workloads experience less than 50% slowdowns, compared to 94%, 87%, and 80% for CXL-D, CXL-A, and CXL-B, respectively. *Many workloads can tolerate CXL latencies.* For example, 60%, 54%, 32% of the workloads on CXL-D, CXL-A, and CXL-B experience less than 10% slowdowns. Correspondingly, 43%, 35%, 22% of the workloads see less than 5% slowdowns. Note that the maximum CXL latency observed is 271ns (CXL-B, excluding CXL-C), demonstrating that many workloads can tolerate CXL access latencies with minimal slowdowns. This category includes certain cloud, HPC, and graph workloads. Consequently, CXL memory can act as a viable drop-in replacement for local DRAM in many real-world applications, with little impact on performance.

The "tail" of CXL slowdowns. The slowdown CDFs in Figure 8b reveal a clear "tail," with 7% of the workloads suffering from significant slowdowns of 1.5-5.8× for CXL-A and CXL-B, primarily due to bandwidth limitations. Interestingly, workloads with high bandwidth requirements do not necessarily experience high slowdowns. In contrast, setups with higher bandwidth capabilities, such as NUMA and CXL-D, exhibit no such tail, with worst case slowdowns limited to 80-90%. For example, in SPEC CPU 2017, four bandwidth-bound workloads – `603.bwaves`, `619.lbm`, `649.fotonik3d`, `654.roms` – require over 24GB/s, exceeding the capacities of CXL-{A, B, C}. As a result, these workloads suffer significant slowdowns (over 50%) compared to NUMA/CXL-D, due to significant device-side queueing delays as the CXL devices become saturated. These four workloads see worse slowdowns under CXL-B and CXL-C because both the latency and bandwidth deteriorate compared to CXL-A.

CXL+NUMA performance. Figure 8c shows workload performance under `CXL+NUMA` is worse than that of 2-hop-NUMA despite the inferior latency and bandwidth of 2-hop-NUMA (`SKX8S-410ns` in Table 1), indicating issues when CXL and NUMA are used together. For example, `520.omnetpp` sees <5% slowdowns under all CXL devices, but experiences an astonishingly high slowdown of 2.9× under `CXL+NUMA`. However, this workload consumes <1GB/s bandwidth (read+write), and is

[3]CXL-C only has 16GB DRAM, limiting evaluations to 60 workloads on it.

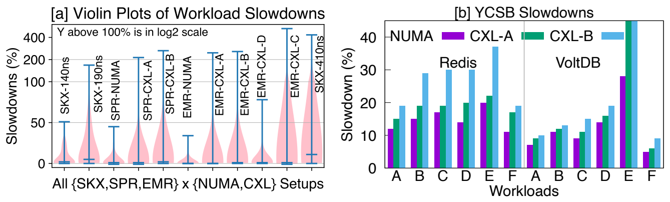

Figure 9. Slowdowns under various setups. *The figure shows the workloads slowdowns in violin plots of all our latency setups.*

neither latency-sensitive nor bandwidth-bound. We confirm the significant slowdown is due to much worse tail latencies under CXL+NUMA, explained next.

Tail-latency impact. `520.omnetpp` performs discrete event simulation of a large Ethernet network. Figure 8d shows the CDFs of sampled memory latencies for it. There is little difference between Local and CXL-A, explaining the small slowdown under CXL-A. However, CXL+NUMA (brown line) exhibits a long tail latency starting around p98 up to 800ns. When we reduce the workload's intensity (by decreasing the number of simulated LANs on backbone switches) to 1/2 and 1/4, the tail latencies consistently improve (two dotted brown lines). The slowdown on CXL+NUMA also significantly decreases from ~290% to ~65% and 58%. *This provides direct evidence that tail latencies are the primary cause of the performance slowdowns.* Similarly, 10 other workloads show negligible slowdowns under CXL but experience slowdowns of 33%-283% under CXL+NUMA.

The closing gap between CXL and NUMA. While current CXL devices do not yet match the performance of NUMA, the gap is narrowing. For example, CXL-D's performance is already close to that of NUMA (Figure 8a, black and green lines). We also test workloads on two CXL-D devices with hardware interleaving, effectively doubling the bandwidth to 104GB/s. As shown in Figure 8f, when CXL bandwidth matches NUMA, performance slowdowns are notably reduced, particularly for workloads that experience high slowdowns with a single CXL-D. Note that the NUMA line in Figure 8f differs from Figure 8a, EMR2S' and CXL-D are a remote host. Figure 8e contrasts workload performance under SPR and EMR. Despite EMR's higher LLC size, slowdown patterns are similar to that of SPR, indicating larger cache is not enough to address CXL-induced latency and bandwidth challenges.

Full latency spectrum (140-410ns). Figure 9a presents violin plots of slowdown for 265 workloads all our setups which consist of different CPU and CXL latency setting combinations. In particular, when latency increases to 410ns (right most plot), the overall slowdowns will be significantly worse than other setups with smaller latencies. However, 16% of the workloads still experience less than 10% slowdowns while 30% less than 50% slowdowns. The rest of the workloads are heavily impacted by both the long latency and limited bandwidth, which is the main reason for the slowdowns. Simply deploying workloads on such (future) CXL setups will be challenging and require careful data placement to curtail the overall performance overhead. Cloud workloads are more sensitive to latencies, Figure 9b shows the slowdown of YCSB workloads measured against Redis and VoltDB, demonstrating the super-linear increasing trend when latency increases.

In summary, the takeaways from our workload study are:

Finding #2:

- Workload performance deteriorates super-linearly with increasing CXL latency; more importantly, the relative slowdowns exceed the rate of the latency increases.
- CXL devices with longer latencies generally achieve lower peak bandwidth (CXL A→B→C), which has a more pronounced impact on bandwidth-bound workloads due to the combined effects of latency and bandwidth.
- CXL devices with worse tail latencies (*e.g.*, CXL-B and CXL-C) experience more significant slowdowns across all evaluated workloads and tail latencies under CXL+NUMA can lead to surprisingly high slowdowns for workloads.
- On a positive note, many workloads can tolerate long CXL latencies (up to 410ns) and thus experience minimal slowdowns, suggesting that CXL could be useful for certain real-world applications, *e.g.*, in pooling scenarios.
- The performance gap between CXL and NUMA diminishes under similar bandwidth capacities, making CXL memory a viable alternative to local/NUMA memory. However, the latency gap remains a challenge for latency-sensitive workloads.

Implication #2: (a) Higher CXL bandwidth will benefit bandwidth-bound workloads, potentially alleviating the 2-6× slowdowns in Figure 8b. Lower latency will enhance the performance of latency-sensitive workloads, such as cloud applications, bringing it closer to NUMA performance. **(b)** When bandwidth is no longer a bottleneck, CXL latency becomes a critical factor for performance, warranting further attention in future CPU/CXL designs and software optimizations. **(c)** However, for bandwidth-bound workloads to fully utilize the combined bandwidth of local and CXL memory, improved software solutions are still required.

Recommendation #2: Workload bandwidth requirements must be carefully assessed when deploying them on CXL devices, as the limited bandwidth of current CXL devices can lead to significant slowdowns under bandwidth pressure.

5 Spa for CXL Slowdown Analysis

5.1 Overview

While quantitative workload characterizations shed light on CXL performance, the root causes of slowdowns remain unclear. We propose Spa[4], a novel root-cause analysis method that uses only 9 CPU counters (Table 2) to model CXL performance with high accuracy (95%) and minimal error (5%) across all workloads (Figure 11a).

[4]Spa stands for **S**tall-based CXL **p**erformance **a**nalysis.

Table 2. CPU counters for Spa. *#c: number of cycles; µops: internal representation of x86 instructions;* `STALLS.SCOREBD` *(P_9) refers to* `RESOURCE_STALLS.SCOREBOARD`*; P_1–P_6 measure stall cycles and P_1–P_9 are available in most Intel processors (SKX, SPR, EMR, GNR, etc.).*

#	Name	Brief Description
P_1	BOUND_ON_LOADS	#c while mem subsys has 1 outstanding load
P_2	BOUND_ON_STORES	#c where the Store Buffer was full
P_3	STALLS_L1D_MISS	#c while L1 miss demand load is outstanding
P_4	STALLS_L2_MISS	#c while L2 miss demand load is outstanding
P_5	STALLS_L3_MISS	#c while L3 miss demand load is outstanding
P_6	RETIRED.STALLS	#c without actually retired µops
P_7	1_PORTS_UTIL	#c when 1 µops was executed on all ports
P_8	2_PORTS_UTIL	#c when 2 µops was executed on all ports
P_9	STALLS.SCOREBD	#c stalled due to serializing operations

The key advantage of Spa lies in its ability to attribute CXL performance slowdowns to specific "sources," such as the CPU cache hierarchy and CXL memory, facilitating a more granular and precise analysis of CXL-induced performance bottlenecks. By isolating the contributions of these components, Spa enables a comprehensive breakdown of workload slowdowns and quantifies their individual impacts on overall performance degradation. Spa can be applied at both the workload level and in a period-based (*i.e.*, time-window) analysis. It tracks CXL performance as workloads progress over time. This dual capability enables CXL slowdown analysis not only for specific workloads but also across temporal phases of workload execution.

5.2 Challenges and Limitations of State-of-the-Art

We aim to quantify the impact of each component to better understand how CXL affects CPU efficiency. For example, instead of the general notion that CPU prefetchers become less effective under CXL's longer latencies [33], we will *measure* CXL's impact on prefetcher performance and *disclose* why it happens. For fine-grained analysis, we need an approach to *capture* the events in the CPU pipeline that lead to performance slowdowns under CXL. While workload slowdowns can be directly measured using application-level metrics, *(a) identifying the underlying CPU events/metrics that can correlate to the slowdowns is challenging. (b) It is even more challenging to establish a precise correlation between workload performance and architecture-level performance metrics,* due to the complexity of the CPU pipeline and workload/CPU interactions. In other words, there is no reliable method to accurately map application slowdowns to system/architecture events, making it an infeasible task with existing approaches.

The Intel Top-down Microarchitecture Analysis (TMA) method [20, 45] is a popular approach that relies on extensive information offered by CPU counters/events to profile code efficiency. It helps identify dominant performance bottlenecks in an application (*e.g.*, whether it's memory-bound) by analyzing execution inefficiencies within the CPU pipeline. However, TMA is insufficient for CXL slowdown analysis for the following reasons.

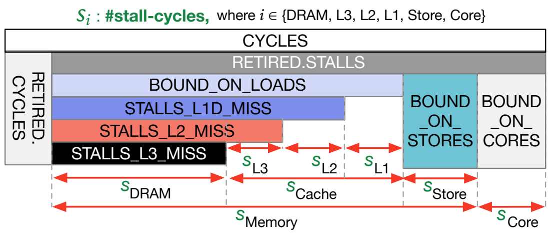

Figure 10. CPU stalls (§5.3). *The figure shows the relationship of various stall-related CPU counters and the originating CPU backend component. Spa relies on differential stalls (Δ) for workload slowdown breakdown. For example, Δs_{L1} is the additional stall cycles on L1 cache (i.e., stalls caused by direct or delayed L1 hits) induced by CXL.*

- TMA does not provide a **differential analysis** to interpret pipeline differences resulting from varying backend memory (*i.e.*, CXL vs. local DRAM).
- TMA is unable to precisely correlate architecture level metrics with workload slowdowns. Its metrics are designed to capture the performance or contention of specific hardware components rather than overall workload behavior.

Consequently, past research heavily relies on simple architectural events/metrics (*e.g.*, LLC miss rate), heuristics or learning methods based on a combination of multiple metrics for performance estimation [34, 41, 47], which suffer from low accuracy, complexity, and lack of interpretability.

5.3 Spa: A Bottom-Up Approach

The key **insight of Spa** *is that drilling down the differential CPU stalls between CXL and local memory can yield accurate CXL slowdown analysis whereas standalone setup (either local or CXL) cannot.* Spa aims to pinpoint the specific stall sources that contribute to CXL-induced slowdowns, bridging the gap between architectural level and workload-level performance. Spa is built on the 9 CPU counters shown in Table 2. Spa sets out to examine the CPU pipeline components involved in memory request processing and analyze the changes induced by CXL on those components during instruction execution. Figure 10 shows an overview of the "sources" of CPU stalls that Spa focuses on. As discussed in §2, processing CXL memory requests requires traversing the memory hierarchy, including L1, L2, LLC, and CXL memory. By evaluating the CPU's efficiency at these key points, we can identify the corresponding slowdowns caused by CXL across workloads. Through detailed offline analysis, we make a few key observations that lead to an accurate slowdown breakdown method that we describe below.

In detail, workload performance slowdowns can be represented using microarchitecture-level performance counters and reasoned about by checking where "stalls" happen in the CPU pipeline. For example, if a workload takes c cycles to complete on local memory and c' on CXL, the slowdown can be denoted as $S = \frac{c'-c}{c} = \frac{\Delta_c}{c}$.

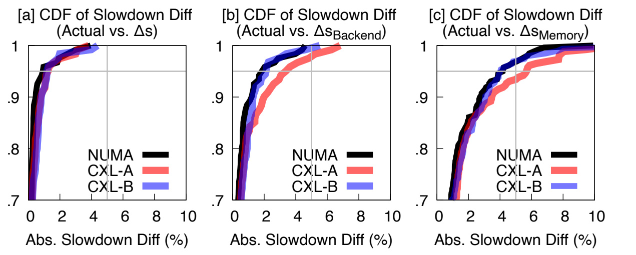

Figure 11. Spa accuracy (§5.3). *The X-axis represents the absolute difference between estimated slowdowns using different kinds of stalls and the actual measured slowdowns for each workload.*

Finding #3: The variance in cycle counts between CXL and local DRAM (*i.e.*, slowdown) primarily stems from *stall cycles* difference (Δs), which mainly arises from the memory subsystem (Δs_{Memory}) in the CPU backend ($\Delta s_{Backend}$).

Based on Figure 10, we can divide the stall cycles into different components, such as core and memory, where each component can be further broken down according to the load/store types as shown in Figure 2c. We calculate the difference (Δ) of a counter between local (P_i) and CXL (P'_i) as $\Delta P_i = P'_i - P_i$. Thus,

$$\Delta s = \Delta P_6 \quad \text{(total \# of additional stalls on CXL)} \tag{1}$$

$$\Delta s_{Backend} = \Delta s_{Core} + \Delta s_{Memory} \quad \text{where,} \tag{2}$$

$$\Delta s_{Core} = \Delta P_7 + \Delta P_8 + \Delta P_9 \tag{3}$$

$$\Delta s_{Memory} = \Delta P_1 + \Delta P_2 \tag{4}$$

For load operations, CPU counter `BOUND_ON_LOADS` (P_1) denotes the number of stalled cycles while there is at least one demand load in the memory subsystem, including CPU cache and (CXL) DRAM. For stores/writes, `BOUND_ON_STORES` (P_2) represents the number of stalled cycles when the Store Buffer is full and there is no outstanding loads. Thereby, the differential Backend stalls are $\Delta P_1 + \Delta P_2$ (Equation (4)).

As such, CXL slowdowns can be estimated as:

$$S = \frac{\Delta c}{c} \approx \frac{\Delta s}{c} \approx \frac{\Delta s_{Backend}}{c} \approx \frac{\Delta s_{Memory}}{c} \tag{5}$$

Purely CPU or frontend-bound workloads are not sensitive to CXL latency due to few CXL accesses, thus experiencing minimal slowdowns (true for our evaluated workloads).

Accuracy. To validate Equation (5), we profile Spa counters in Table 2 for each workload and use them to estimate the workload slowdowns. We compare them with the actually observed workload slowdowns using application-level metrics (*e.g.*, time, throughput). Figure 11 presents the CDF plots of the absolute difference between the actual slowdown and differential-stall based slowdown estimations, which indicates the inaccuracies. We show the results for NUMA, CXL-A, and CXL-B. Figure 11a shows that stall cycles difference can accurately represent the slowdowns for 100% of workloads, with the absolute difference within 5% (and 98% accuracy under 2% difference). Figure 11b shows that backend-stall-based slowdowns can accurately represent the slowdowns for 96% of workloads, with the absolute difference within 5%, because CXL introduces minimal frontend stalls differences. In Figure 11c, by comparing the slowdown estimation based on the memory subsystem stalls (Δs_{Memory}), we observe very low inaccuracies – within 5% for over 95% of workloads. Therefore, CXL-induced stall cycle difference (Δs_{Memory}) can effectively represent the slowdown.

Reasoning. The CPU pipeline is divided into two parts: the frontend and the backend. In the frontend, instructions are fetched and decoded, while in the backend, they are executed out-of-order (OoO), through the Scoreboard scheduling via Reorder Buffer [16]. Stalled cycles can occur due to stalls in either the frontend, the backend, or both. In our analysis, we only consider the difference of stalls between local and CXL memory, rather than looking at the absolute number of stalls in either setup (local or CXL) because the absolute stalls cannot be used for CXL slowdown analysis. Throughout our evaluations of all the workloads, we find that delta of frontend stalls are negligible because modern CPU instruction caches are efficient and large enough to fetch and decode instructions without being affected by CXL delays. Note that many of the tested workloads are indeed frontend-bound (>30%). Therefore, it is primarily stalls in the CPU backend that are impacted by CXL, as shown in Figure 11b.

5.4 Spa-based Slowdown Breakdown

Additional stalls on Core (Δs_{Core}) under CXL. Non-load/store instructions executed on the cores are not affected by CXL's longer latency. The main pressure caused by CXL latency on the CPU cores is mostly from handling data dependency for OoO executions. P_8 and P_9 collect the number of cycles when only 1 and 2 μops are executed on all execution ports (*e.g.*, ALU). Meanwhile, CXL's longer latency can cause more stalls on the scoreboard when the scoreboard deals with intensive data serializing operations. However, all the impact from the longer memory latency on Core is relatively small, when comparing Figure 11b with Figure 11c.

Breaking down the slowdown. On Intel platforms, the stalls on the store buffer, L1, L2, LLC, and (CXL) DRAM represent exclusive events that sum up to the total backend-memory stall cycles as shown in Figure 10 (also see Figure 4 in [45]). Let s be the number of stall cycles, we have:

$$s = s_{store} + s_{L1} + s_{L2} + s_{L3} + s_{DRAM} \tag{6}$$

where, s_{store}=P_2, s_{L1}=P_1-P_3, s_{L2}=P_3-P_4, s_{L3}=P_4-P_5. With this, when calculating the difference of above stall cycles between local DRAM and CXL, we have:

$$\Delta s = \Delta s_{store} + \Delta s_{L1} + \Delta s_{L2} + \Delta s_{L3} + \Delta s_{DRAM} \tag{7}$$

Here, Δs_{L1} denotes the difference (Δ) of stall cycles on L1 on local and CXL DRAM. Correspondingly, by dividing each item with total cycle-count (c), the overall slowdown (S) can be represented as the combined slowdowns from the five sources as follows:

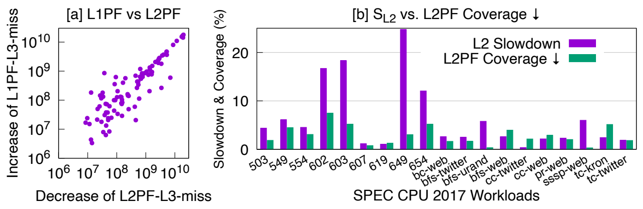

Figure 12. Cache slowdown vs. L2PF coverage decrease. *Cache slowdown is correlated with L2 prefetcher coverage decrease.*

$$S \approx S_{store} + S_{L1} + S_{L2} + S_{L3} + S_{DRAM} \quad (8)$$

Above, each component-wise slowdown is calculated as the delta of stall cycles on the specific component, *e.g.*, slowdown due to L1 cache access is Δ of stalled cycles on L1, denominated by the total cycle count (c), *i.e.*, $S_{L1} = \Delta s_{L1}/c$.

DRAM (Demand Load) Slowdown (S_{DRAM}). We use the increase in LLC stall cycles, as a primary indicator of CXL slowdown from DRAM. These misses denote *demand read misses*, excluding RFO and prefetch requests. On Intel platforms, they are characterized as cycles stalled while LLC demand read misses are unresolved. Hence, their change suggests performance deterioration originating from DRAM, including the (CXL) memory controller.

Store Slowdown (S_{store}). We use the increase of cycles bound on full store buffer to gauge store operation slowdown. Incoming `store` requests queued in the store buffer are dequeued upon completion. Some writes issue RFO requests before execution. If the store buffer fills up, these RFOs would hinder load efficiency, causing CPU stalls.

Cache Slowdown (S_{cache}). While DRAM and store slowdowns are relatively straightforward to understand, cache slowdowns are more complex. In this section, we discuss our key findings on how CXL can degrade CPU cache efficiency. Cache slowdown (S_{L1}+S_{L2}+S_{L3}) indicates stall cycle increase on various cache levels (L1, L2, and LLC), as it takes longer to prefetch from CXL memory. Similarly, they can be measured using the corresponding stall cycles counters in Figure 10. Below we describe our findings to reason about cache slowdowns on CXL through offline analysis. On SKX, most cache slowdown occurs in L2 due to a significant rise in stall cycles for L1 load misses with CXL. Conversely, on SPR/EMR, LLC experiences the bulk of slowdown, with a notable increase in stall cycles for L2 load misses with CXL.

Finding #4:

- Cache slowdown under CXL is due to reduced prefetch efficiency. To validate this, we disable all the hardware prefetchers (L1 and L2) and measure workload slowdowns. With prefetchers off, we found virtually no stall cycles on cache (*i.e.*, $S_{L1} = S_{L2} = S_{L3} = 0$).
- Through our extensive offline analysis, we find CXL's relatively longer latency negatively impacts L2 prefetcher's timeliness (*i.e.*, L2 prefetches are delayed and take longer to arrive). This reduces L2 prefetcher's coverage of both demand reads and L1 prefetch. L1 prefetches would either miss entirely in L2 or at best, they would hit on a pending L2 prefetch in L2. Consequently, CXL also negatively impacts L1 prefetcher's timeliness. Loads that would have otherwise hit in the cache if L1 prefetches were timely, now are delayed. Consequently, overall prefetch efficiency suffers and stalls on caches increase.

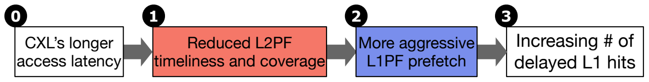
0
CXL's longer access latency
1
Reduced L2PF timeliness and coverage
2
More aggressive L1PF prefetch
3
Increasing # of delayed L1 hits

Figure 13. Cache slowdown flow under CXL. *Illustration of CXL cache slowdowns based on our offline analysis.*

Figure 13 summarizes the cache slowdowns under CXL. The increased latency of CXL initially reduces the efficiency of L2 prefetchers, resulting in less useful data being available in the L2 cache. Consequently, L1 prefetchers must retrieve more data from the LLC or CXL DRAM due to L2 misses. Additionally, CXL impacts L1 prefetch efficiency, as the delayed data retrieval leads to delayed L1 hits, further contributing to L1 cache slowdowns.

Reasoning. L1 prefetchers retrieve data from L2, LLC, or DRAM into the line-fill buffer (LFB, Figure 2a). If the requested data is absent in L2, the request is forwarded to off-core resources (LLC or DRAM/CXL). Ideally, breaking down L1/L2 prefetch requests would enable a more comprehensive analysis to fully understand cache slowdowns. However, Intel CPUs lack the necessary performance counters to capture all L1/L2 prefetcher interactions with CXL, such as the number of L1 prefetch requests that hit L2 (L1PF-L2-hit). This limitation makes it difficult to precisely examine hardware prefetcher behaviors. Instead, we leverage the available performance counters that measure **L1PF-L3-miss** and **L2PF-L3-miss** to evaluate L1/L2 prefetcher performance, particularly under CXL's longer memory latencies, where cache misses are more pronounced. Both L1PF-L3-miss and L2PF-L3-miss are derived from raw counters. Specifically, Intel provides performance counters that track the number of prefetch requests issued by L1/L2 prefetchers that miss in L2 (fetching from either LLC or DRAM) and separate counters for prefetch requests that hit in L3/LLC. Overall, we observe a 2-38% reduction in L2PF-L3-miss for CXL compared to local DRAM, aligning with the L2 cache slowdown results presented in Figure 12b.

In particular, we found a decrease in L2 prefetch requests that miss the L3/LLC cache (L2PF-L3-miss) under CXL, while L1 prefetch misses (L1PF-L3-miss) increase. Notably, the increase in L1PF-L3-miss nearly matches the decrease in L2PF-L3-miss, as shown in Figure 12a, with no change in L2PF-L3-hit. This suggests that the L2 prefetcher is less efficient in fetching data from CXL compared to local DRAM, resulting in more L1 prefetches bypassing L2 and fetching data directly from CXL. The strong linear relationship between

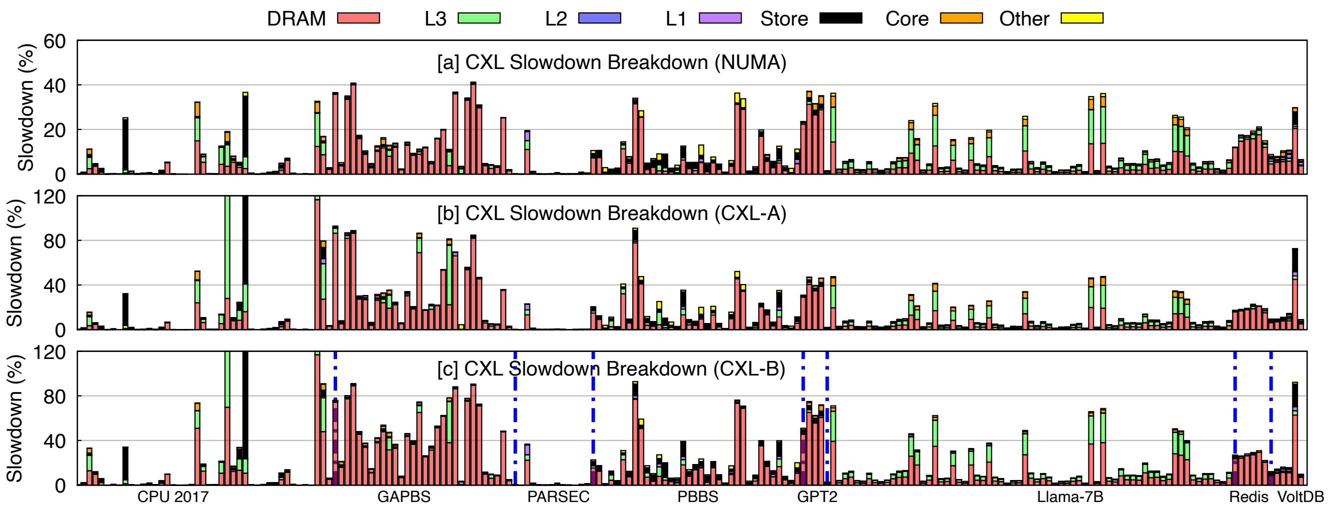

Figure 14. CXL breakdown (§5.5). *This figure shows* Spa*-based CXL slowdown breakdown for NUMA and CXL-A on EMR, attributing the slowdowns to various sources (DRAM, L1–LLC, Store, Core). CXL-C and CXL-D results exhibit similar patterns (not shown).*

L2PF-L3-miss decrease and L1PF-L3-miss increase (almost $y = x$, Pearson coefficient 0.99) confirms this. Further analysis shows that workloads experiencing cache slowdowns on CXL consistently exhibit reduced L2 prefetch coverage, as demonstrated in Figure 12b.

Disabling L1 and L2 prefetchers results in data that would otherwise be prefetched being retrieved as LLC misses. Consequently, the slowdown initially seen in the caches is transferred to DRAM slowdowns. Enabling prefetchers significantly improves workload performance under both local and CXL setups. For example, we observed a 50% performance drop in workload `603.bwaves` and a 10% drop for `bc-kron` graph workload when L1 and L2 prefetchers are disabled. While prefetchers alleviate long memory latencies by reducing on-demand LLC misses, they are less effective under CXL, as captured by the cache slowdown.

5.5 Workload Slowdown Diversity

Figure 14 shows the overall and detailed breakdown of CXL slowdowns for each workload. The "Other" category represents slowdown contributions not captured by Spa.

The sources of CXL slowdown vary significantly across workloads. For instance, in SPEC workloads like `519.lbm`, the majority of the slowdown is due to stalls in the CPU's store buffer, suggesting a high volume of RFO requests and insufficient store buffer entries. These findings are supported by high UPI non-data traffic and high write bandwidth. In contrast, workloads like `649.fotonik3d` experience significant slowdown from cache-related issues.

For GAPBS workloads, the primary source of slowdown stems from DRAM, specifically stalls by demand reads. Only a few, such as `pr-kron` and `pr-twitter`, experience cache-related slowdowns. Many Llama workloads exhibit slowdowns originating from LLC. Similarly, cloud workloads like Redis/VoltDB are predominantly impacted by DRAM slowdowns. In ML workloads such as DLRM and GPT-2, DRAM slowdowns account for 90% of the overall slowdown. Figure 15 shows the CDFs of slowdowns caused by various components for all workloads. Notably, at least 15% of workloads experience 5% or more cache slowdown under CXL, indicating reduced prefetcher efficiency. Additionally, at least 40% of workloads see 5% or more slowdown due to demand reads.

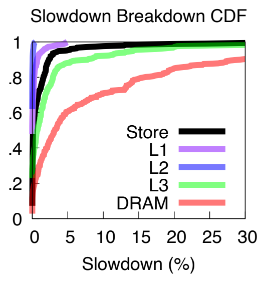

Figure 15. Slowdown.

Reasons behind workload slowdowns vary significantly. For instance, `503.bwaves` and `605.mcf` demonstrate similar slowdowns. However, almost all slowdown in `605.mcf` is from LLC misses, while for `503.bwaves`, prefetching dominate the performance slowdown. This underscores one of the advantages of the breakdown method. To sum up, Spa can capture, explain, and dissect slowdowns.

5.6 Period-based Slowdown Analysis

Spa at the workload level falls short of capturing workload dynamism and the associated CXL impact over time. Modern memory-intensive workloads often exhibit bursty behavior and experience periodic performance fluctuations. Therefore, it is essential to analyze workload slowdowns over various time periods during execution to understand the evolving causes. To this end, we enhance Spa to support period-based slowdown analysis throughout the workload's lifetime, *e.g.*, every 1 billion (B) instruction "period."

Challenges: Measuring period-based slowdowns is nontrivial as the same set of instructions will take different amount of time to execute in local DRAM and CXL memory while existing profiling tools only support time-based sampling (*e.g.*, measuring performance counters every 1ms), but cannot deal with the issue of **runtime scaling**.

Our solution: We introduce an approach to convert time-

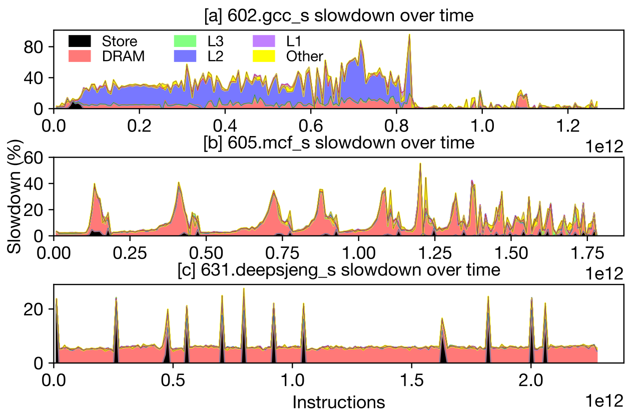

Figure 16. Period-based slowdown breakdown. *This figure shows the time-series CXL slowdown for workload 602, 605 and 631.*

based sampling data into a period-based slowdown analysis. Since the total number of (retired) instructions for a workload remains constant whether running on DRAM or CXL memory, our method aggregates multiple time-based samples (*i.e.*, counters in Table 2) over 1ms intervals and then align them with the target analysis period that lasts much longer (*e.g.*, every 1B instructions, seconds). Partial time-based sampling results are proportionally adjusted, assuming smooth counter progression within the 1ms sampling interval. As the time-based sampling intervals are significantly shorter than the target analysis period, errors are minimized by proportional offsets during the final interval. We apply this approach to time-series performance counter data collected periodically from both DRAM and CXL setups, yielding instruction-interval-based breakdowns. Using this method, we perform fine-grained workload analysis and present example findings in Figure 16.

For instance, consider workload (`602.gcc`), which experiences significant CXL slowdowns during the first two-thirds of its execution. The average slowdown within this period exceeds 30%, whereas the overall average slowdown across the workload hovers around 20%. While the average slowdown for the entire workload fails to pinpoint critical time periods with high slowdown, our method enables the presentation of slowdown for each time period, thereby revealing critical segments affected by substantial performance degradation. Concurrently, the time series analysis highlights the fluctuation of various slowdown components. Workloads `605.mcf` and `631.deepsjeng` exhibit comparable intensity in overall slowdown, yet their respective contributions to slowdown during execution fluctuate differently.

Finding #5: Temporal slowdown variation: Different workloads may exhibit distinct CXL slowdown fluctuations over time, even if their average slowdowns are similar at workload-level. Period-based slowdown analysis helps uncover these runtime performance variations. By identifying less-affected periods, resource utilizations could be optimized, benefiting other workloads under co-location.

Recommendation #3: Period-based slowdown analysis offers valuable insights for program optimization by helping programmers identify which instruction periods are most susceptible to CXL slowdowns, enabling more precise and targeted optimization.

5.7 Spa Use Cases

Performance debugging. As shown in previous sections, Spa serves as a valuable complement to existing profilers, offering deeper insights into CXL performance profiling and debugging. By identifying root causes of performance slowdowns and visualizing their impact (as in Figure 16), Spa enables users to analyze workload behavior under CXL and optimize application performance accordingly.

Performance tuning. Spa facilitates workload optimization by enabling informed memory placement based on slowdown analysis. For example, to mitigate the slowdown bursts observed in `605.mcf` (Figure 16b), we first identify memory accesses during bursty periods (*e.g.*, exceeding 10%) using binary instrumentation via Intel Pin. Next, we pinpoint the source code responsible for high slowdowns using `addr2line`. Our analysis reveals that two performance-critical objects, each 2GB in size, are contributing to the slowdown. By relocating these objects to local DRAM, we successfully reduce the overall slowdown from 13% to 2%. This approach demonstrates how Spa can effectively guide memory placement between local DRAM and CXL, making it a valuable tool for resource provisioning in memory pooling and tiering systems without violating SLOs [23, 34].

Performance prediction and metric. Spa serves as a foundation for accurate predictive models applicable across various scenarios, as detailed in our technical report [35]. Spa-based models provide a powerful framework for analyzing and predicting workload performance in complex memory configurations, such as tiered memory in both online and offline settings. As a performance metric, Spa offers a more effective alternative to conventional metrics like LLC misses. By directly measuring performance losses through stall cycles, Spa enables smarter tiering policy designs that improve both system performance and memory utilization.

6 Conclusion

As CXL remains a relatively new technology, the research community has yet to fully understand its performance implications. Our systematic, large-scale characterization and analysis provide key findings and actionable insights. By leveraging unique analytical perspectives – including tail latencies, CPU tolerance to CXL latencies, and benchmarking many real-world workloads – Melody offers a comprehensive understanding of CXL performance. We hope that Melody findings, tools, datasets, and Spa will spur further research to fully comprehend and effectively manage the complexities of CXL-induced performance dynamics.

7 Acknowledgments

We thank Nicolai Oswald (our shepherd) and the anonymous reviewers for their constructive feedback, which enhanced this paper. We also thank Brian Hirano and Ming Liu for their input on an early draft, as well as Shoaib A. Qazi and Hansen Idden for their assistance. This research was partially supported by an NSF CAREER Award (CNS-2339901), NSF grant CNS-2312785, and funding and equipment support from Microsoft, Samsung, Intel, and 4-VA.

References

[1] A Benchmark Suite for Cloud Services. https://github.com/parsa-epfl/cloudsuite.

[2] Compute Express Link. https://www.computeexpresslink.org.

[3] CXL Memory eXpander Controller (MXC). https://www.montage-tech.com/MXC.

[4] CZ120 Memory Expansion Module. https://www.micron.com/products/memory/cxl-memory.

[5] GPT-2. https://en.wikipedia.org/wiki/GPT-2.

[6] Intel Memory Latency Checker (Intel MLC). https://www.intel.com/content/www/us/en/download/736633/intel-memory-latency-checker-intel-mlc.html.

[7] JEDEC Memory Module Reference Base Standard - for Compute Express Link (CXL). https://www.jedec.org/standards-documents/docs/jesd319.

[8] Leo CXL Smart Memory Controllers. https://www.asteralabs.com/products/leo/leo-cxl-memory-connectivity-controllers/.

[9] LLM Inference in C/C++. https://github.com/ggerganov/llama.cpp.

[10] PCIe 6.0 Will Run So Hot That It Needs Thermal Throttling. https://tinyurl.com/pciegen6-2.

[11] PCIe 6.0's Thermal Throttling Plans Could Slam Brakes on Performance. https://tinyurl.com/pciegen6-1.

[12] Phoronix. https://www.phoronix.com/.

[13] Redis. https://redis.io.

[14] Reference Implementations of MLPerf Inference Benchmarks. https://github.com/mlperf.

[15] Samsung Unveils CXL Memory Module Box: Up to 16 TB at 60 GB/s. https://www.anandtech.com/show/21333/samsung-unveils-cxl-memory-module-box-up-to-16-tb-at-60-gbs.

[16] Scoreboarding. https://en.wikipedia.org/wiki/Scoreboarding.

[17] SK hynix CXL 2.0 Memory Expansion Modules Launched with 96GB of DDR5. https://www.servethehome.com/sk-hynix-cxl-2-0-memory-expansion-modules-launched-with-96gb-of-ddr5/.

[18] SPEC CPU 2017. https://www.spec.org/cpu2017.

[19] The PBBS Benchmark Suite (V2). https://cmuparlay.github.io/pbbsbench/.

[20] Top-down Microarchitecture Analysis Method. https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html.

[21] VoltDB. https://www.voltdb.com.

[22] GAP Benchmark Suite. https://github.com/sbeamer/gapbs.git, 2021.

[23] Daniel S. Berger, Daniel Ernst, Huaicheng Li, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Lisa Hsu, Ishwar Agarwal, Mark D. Hill, and Ricardo Bianchini. Design Tradeoffs in CXL-Based Memory Pools for Cloud Platforms. *IEEE Micro Special Issue on Emerging System Interconnects*, 43(2), 2023.

[24] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[25] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, Optimization. In *Proceedings of the 2016 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2016.

[26] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. Quantifying the Performance Impact of Memory Latency and Bandwidth for Big Data Workloads. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2015.

[27] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.*, 56(11), July 2024.

[28] Pouya Esmaili-Dokht, Francesco Sgherzi, Valeria Soldera Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Momeni, Adria Armejach, Estanislao Mercadal, German Llort, Petar Radojkovic, Miquel Moreto, Judit Gimenez, Xavier Martorell, Eduard Ayguade, Jesus Labarta, Emanuele Confalonieri, Rishabh Dubey, and Jason Adlard. A Mess of Memory System Benchmarking, Simulation and Application Profiling. In *57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-57)*, 2024.

[29] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. In *Proceedings of the 2019 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2019.

[30] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, 2010.

[31] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[32] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[33] Philip Levis, Kun Lin, and Amy Tai. A Case Against CXL Memory Pooling. In *The 22nd ACM Workshop on Hot Topics in Networks (HotNets '23)*, 2023.

[34] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 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)*, 2023.

[35] Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S. Berger, and Huaicheng Li. Dissecting CXL Memory Performance at Scale: Analysis, Modeling, and Optimization. https://arxiv.org/abs/2409.14317, 2024.

[36] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 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 (ASPLOS)*, 2023.

[37] Vinicius Petrucci, Eishan Mirakhur, Nikesh Agarwal, Su Wei Lim, Vishal Tanna, Rita Gupta, and Mahesh Wagh. CXL Memory Expansion: A Closer Look on Actual Platform. https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansion-a-close-look-on-actual-platform.pdf.

[38] Christian Pinto, Dimitris Syrivelis, Michele Gazzetti, Panos Koutsovasilis, Andrea Reale, Kostas Katrinis, and Peter Hofstee. ThymesisFlow: A Software-Defined, HW/SW Co-Designed Interconnect Stack for Rack-Scale Memory Disaggregation. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*, 2020.

[39] Huanxing Shen and Cong Li. Runtime Estimation of Application Memory Latency for Performance Analysis and Optimization. In *The International Symposium on Memory Systems (MEMSYS)*, 2020.

[40] Shigeru Shiratake. Scaling and Performance Challenges of Future DRAM. In *IEEE International Memory Workshop (IMW)*, 2020.

[41] Yan Sun, Yifan Yuan, Zeduo Yu, 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. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-56)*, 2023.

[42] 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. Exploring Performance and Cost Optimization with ASIC-Based CXL Memory. In *Proceedings of the 2024 EuroSys Conference (EuroSys)*, 2024.

[43] Jacob Wahlgren, Gabin Schieffer, Maya Gokhale, and Ivy Peng. A Quantitative Approach for Adopting Disaggregated Memory in HPC Systems. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2023.

[44] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. Designing Cloud Servers for Lower Carbon. In *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[45] Ahmad Yasin. A Top-Down Method for Performance Analysis and Counters Architecture. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[46] Li Yi, Cong Li, and Jianmei Guo. CPI for Runtime Performance Measurement: The Good, the Bad, and the Ugly. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2020.

[47] 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. Managing Memory Tiers with CXL in Virtualized Environments. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.