Dissecting CXL Memory Performance at Scale: Analysis, Modeling, and Optimization

Jinshu Liu, Hamid Hadian, Hanchen Xu, Daniel S. Berger†, Huaicheng Li
Email: {jinshu,huaicheng}@vt.edu
Virginia Tech †Microsoft

Abstract

Compute Express Link (CXL) is a promising interconnect technology that enables system memory expansion, but it comes at the cost of long latencies and low bandwidth compared to socket-local memory. To fully understand the performance potential of CXL and mitigate its high latency overhead, a detailed characterization of CXL performance is crucial to guide the modeling and optimization of CXL memory systems.

We present SupMario, a characterization framework designed to thoroughly analyze, model, and optimize CXL memory performance. SupMario is based on extensive evaluation of 265 workloads spanning 4 real CXL devices within 7 memory latency configurations across 4 processor platforms. SupMario uncovers many key insights, including detailed workload performance at sub-µs memory latencies (140-410 ns), CXL tail latencies, CPU tolerance to CXL latencies, CXL performance root-cause analysis and precise performance prediction models. In particular, SupMario performance models rely solely on 12 CPU performance counters and accurately fit over 99% and 91%-94% workloads with a 10% misprediction target for NUMA and CXL memory, respectively.

We demonstrate the practical utility of SupMario characterization findings, models, and insights by applying them to popular CXL memory management schemes, such as page interleaving and tiering policies, to identify system inefficiencies during runtime. We introduce a novel "best-shot" page interleaving policy and a regulated page tiering policy (Alto) tailored for memory bandwidth- and latency-sensitive workloads. In bandwidth bound scenarios, our "best-shot" interleaving, guided by our novel performance prediction model, achieves close-to optimal scenarios by exploiting the aggregate system and CXL/NUMA memory bandwidth. For latency sensitive workloads, Alto, driven by our key insight of utilizing "amortized" memory latency to regulate unnecessary page migrations, achieves up to 177% improvement over state-of-the-art memory tiering systems like TPP, as demonstrated through extensive evaluation with 8 real-world applications.

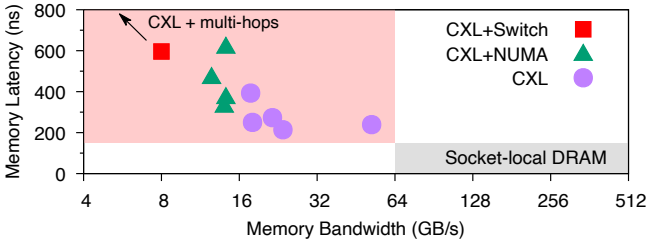

Figure 1: **CXL latency and bandwidth heterogeneity.**

1 Introduction

The demand for increased memory capacity is rapidly rising, driven by the growing requirements of data-intensive applications [43]. The surge is further compounded by DRAM scaling challenges [47]. Emerging interconnects like Compute Express Link (CXL) holds the promise of both scale-up and scale-out coherent memory expansion at the server/rack levels [6, 41, 42]. Memory vendors have introduced CXL memory expanders [4, 5, 10, 21], facilitating access to significantly larger amounts of DRAM than previously feasible. For instance, Samsung's CXL Memory Module - Box (CMM-B) [21] offers 16TB of DRAM with 8 CXL devices.

Memory performance is key to system performance. However, CXL memory expansion introduces higher access latencies compared to traditional socket-local DRAM configurations. Figure 1 illustrates the substantial heterogeneity in CXL latency and bandwidth, as measured across various CXL devices within our platform (Table 1) and from public sources[1] [21, 22]. Furthermore, CXL devices can exhibit varying performance characteristics. The variability in latency and bandwidth arises from varying interconnection topologies and vendor optimizations. For instance, the latencies of locally-attached CXL range from ∼200-400ns, slightly exceeding cross-socket/NUMA latency. Accessing CXL from a remote socket results in increased latency and diminished bandwidth (`CXL+NUMA`). The incorporation of a CXL switch to extend connectivity will introduce additional latencies (`CXL+Switch`), even elevating latency to approximately 600ns. In the future, with CXL potentially involving

[1] `CXL+Switch` data is from [21], bandwidth averaged for 1 CXL device.

multiple routing hops and its use with slow memory media (*e.g.*, Flash) [20], latency is projected to increase to µs-level.

The current CPU architecture and memory hierarchy are tailored for typical 1-2 socket systems, offering ∼100ns latency and 100s of GB/s bandwidth. However, the performance implications of emerging CXL memory technology remain uncertain. Currently, there is a lack of research exploring detailed CXL characteristics and its impact on memory-intensive workloads at large-scale. Conducting a thorough characterization is crucial to provide valuable insights for the imminent CXL deployment in production systems and software/hardware memory management.

In particular, how do CXL devices vary from each other in terms of detailed performance characteristics? How does CXL's long latency impact CPU efficiency and workload performance? What are the root causes? Addressing these questions requires a deep understanding of the dynamic nature of CXL's performance characteristics, which span a spectrum rather than adhering to fixed, static values of latency and bandwidth. While previous studies [41, 44, 48–50] provide valuable insights into CXL performance impact, they are primarily done at a coarse-grained level, overlooking critical aspects such as CXL performance stability (*i.e.*, tail latencies), CPU tolerance to long CXL latencies, CXL's architectural implications and performance predictability.

We present **SupMario**, a comprehensive characterization framework for large-scale CXL performance profiling, analysis, modeling, and optimizations. Our goals are:

(1) Understanding CXL latency and throughput implications. How (much) does CXL impact workload performance? What are the root causes? And how to reason about it systematically? Can workloads benefit from the higher aggregate memory bandwidth by splitting the dataset between local and CXL memory and how? We conduct a large-scale performance study of the characteristics of 4 CXL devices and assess 265 workloads across 7 memory latency configurations ranging from 140-410 ns on 4 processor platforms. This study provides a quantitative analysis of CXL performance at scale, uncovering new findings and insights that would not have been possible without a large-scale approach.

(2) Memory performance modeling. Can lightweight models reliably predict workload performance in CXL-enabled environments? Through an in-depth root-cause analysis complementing our characterization findings, we delve into CXL implications on CPU efficiency and develop novel linear models for workload performance prediction under CXL. Our models are based on novel combinations of solely 12 CPU performance counters but can work surprisingly well. We emphasize that our accurate prediction models represent a significant advancement in enhancing the **observability and predictability of memory system performance**. They are *simple, easy-to-use, explainable, general*, and can serve as fundamental performance metrics which we believe can potentially enable many use cases.

(3) Memory performance optimization. What are the limitations of existing memory policies in managing CXL memory, and how can we leverage CXL characteristics to design better memory management policies? We show that SupMario's approach can be used to quantitatively analyze the inefficiencies of complex memory policies in managing CXL memory. Additionally, we can leverage insights from SupMario to develop enhanced memory management strategies. We apply SupMario's characterization techniques and prediction models to memory tiering [2] and interleaving [26]. Our experiments demonstrate the effectiveness and broad applicability of SupMario's insights in identifying system inefficiencies and enhancing the observability of complex memory systems. More importantly, we introduce SupMario-augmented interleaving and tiering policies, which lead to significant performance improvements compared to state-of-the-art. In summary, our key contributions are:

(1) SupMario, the largest-scale CXL performance study, to the best of our knowledge, characterizing 265 workloads under 4 real CXL devices across 7 memory latency configurations on 4 processors, detailing many new findings about workload performance under sub-µs memory latencies, CXL device performance (such as latency stability) and deep-dive analysis of CXL and CPU interactions across workloads and setups.

(2) A novel root-cause analysis approach based on CPU stall cycles for workload performance dissection under CXL, identifying and quantifying various sources of CXL-induced performance degradations in the CPU.

(3) A linear performance prediction model for both latency and bandwidth sensitive scenarios that are *workload-independent and robust* (validated under multiple CXL and processor platforms and various memory policies), *simple and lightweight* (using only 12 CPU performance counters), *accurate* (for both NUMA and CXL memory), and *explainable* (from root-cause performance breakdown analysis).

(4) A "best-shot" page interleaving policy for bandwidth-bound workloads to effectively utilize both system and CXL bandwidth simultaneously, achieving near-ideal bandwidth improvements[2].

(5) Alto, a memory tiering policy based on a core insight of "amortized" memory access latency by incorporating both memory-level-parallelism and access latencies to precisely capture the impact of page migrations to workload performance. By minimizing unnecessary page migrations and reducing the associated overhead, Alto achieves up to 177% improvement compared to TPP [42], a popular CXL memory tiering solution.

[2]Calculated as CXL bandwidth over system socket-local DRAM bandwidth, *i.e.*, $\texttt{BW}_{\texttt{CXL}}/\texttt{BW}_{\texttt{DRAM}}$).

2 Background and Motivation

Below we present CXL background on the protocol, CPU-CXL interactions, memory profiling, and CXL memory management policies.

CXL for memory expansion. CXL [3] is an emerging cache coherent interconnect built atop PCIe. It enables many potential use cases, such as memory expansion, pooling, and sharing. CXL memory seamlessly integrates into systems as cacheable, byte-addressable memory within a *zero-core NUMA* (zNUMA) node (*i.e.*, CPU-less NUMA) [41]. Thus, applications can simply treat it as a slower-tier of memory compared to local DRAM. Although CXL outperforms PCIe in speed due to tailored transaction and link protocols, it is commonly perceived that its latency is comparable or slightly worse than that of one NUMA hop [17]. Moreover, CXL can increase system bandwidth, potentially benefiting bandwidth-bound workloads. Despite the rollout of CXL products in the last three years, there remains a lack of in-depth studies to comprehensively understand their performance implications, which motivates our work.

CXL request processing. In a conventional pyramid-shaped memory hierarchy [12] with L1, L2, and L3/LLC caches, if a memory request (*e.g.*, reading 64B of data) is not satisfied by the L1–L3 caches due to cache misses, the request is forwarded to the CXL memory controller (MC) via the CXL link. Once CXL memory returns the requested data, L1–L3 caches are updated to serve future requests more efficiently. At a high level, the CPU's request processing flow remains the same for both local DRAM and CXL [33]. However, the use of different buses (DIMM vs. CXL/PCIe) and MCs (on-CPU integrated, *i.e.*, IMC vs. third-party) affects the efficiency of CPU cache hierarchy.

The `load/store` interface is used for a CPU to communicate with integrated or CXL MC to perform memory operations. The CPU issues two types of `load` requests: on-demand and prefetching read operations. On-demand loads are memory read operations where the CPU requests data from (CXL) memory only when it is needed for computation while prefetching loads are predictive reads (directed by the hardware prefetchers) in advance. The CPU issues store requests to write data to memory. To maintain cache coherence, if the CPU wants to modify a cacheline, it needs to first send a read-for-ownership (RFO) request to gain exclusive access to the cacheline by asking the other cores to invalidate their copies of the cacheline and/or load the cacheline from (CXL) memory. Thus, the (CXL) MC needs to handle three types of memory reads: on-demand, prefetching, and RFO. We will later show that differtiating the three types of memory reads is crucial for understanding CXL's performance implications (more in §4).

For example, CPUs heavily rely on hardware prefetchers to minimize potential pipeline [8] stalls caused by the longer access latency of (CXL) DRAM compared to L1–L3 caches. The pipeline refers to the multiple instruction processing stages for concurrent instruction executions, which helps improve CPU speed. However, the increased CXL access latency can lead to delayed request prefetching, causing the CPU pipeline to stall for a longer period (*i.e.*, waiting for data to arrive, more in §4). This results in degraded workload performance under CXL.

CXL profiling and profile-guided optimizations. Modern CPUs offer robust profiling capabilities through hardware counters/events sampling for top-down microarchitecture analysis (TMA) [54]. This technique has been integrated into widely used profilers, *e.g.*, Linux `perf`. TMA allows users to pinpoint CPU inefficiencies with well-defined metrics. For example, DRAM-bound metric measures how often CPU was stalled on DRAM. As modern data-intensive workloads becomes increasingly memory-bound, they can lead to significant stalled CPU cycles [37]. This approach is important for understanding performance issues that arise from the inherent memory access patterns of these workloads.

Leveraging such information to inform system optimizations is a well-established practice [39, 42, 45]. One common strategy involves utilizing hardware performance counters/events, either individually or in combination, within heuristic or ML algorithms as performance predictors. However, it remains a challenge to define accurate performance metrics that can capture complex system behaviors. There are two limitations with existing approaches: accuracy and complexity. Many widely used performance indicators, such as `LLC-miss`, are inaccurate. And ML methods introduce high computational overhead to be useful for scenarios with tight time constraints of 100s of ns. Thus, TMA is mainly used for offline workload analysis. We will address this with a clever combination of multiple performance counters to serve as reliable performance predictors (§5) and use them online for system optimizations (§6).

CXL memory management. Utilizing CXL as regular DRAM can lead to suboptimal performance due to CXL's longer latency and/or relatively smaller bandwidth. There are two popular approaches to address this challenge: (NUMA) page interleaving and memory tiering. Page interleaving involves distributing page allocations across NUMA nodes in round-robin to maximize bandwidth usage [13]. In contrast, tiering aims to minimize CXL latency impact by prioritizing local DRAM for most-frequently accessed pages via proactive page migrations. While interleaving and tiering have been studied across various heterogeneous memory contexts, including CXL, persistent memory, and disaggregated memory [38, 39, 42, 45, 56], fundamental gaps remain in effective tiering policy designs.

In the rest of the paper, we present characterizations in §3 and §4, CXL/NUMA performance models in §5, system optimizations for interleaving and tiering in §6, and conclude in §9 followed by discussion and related work in §7 and §8.

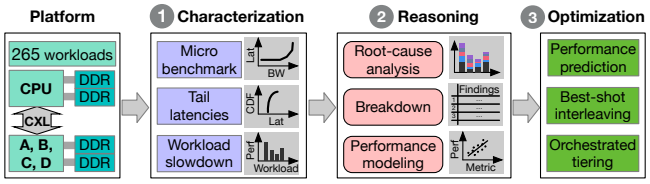

Figure 2: **Overview.** *Our in-depth and at-scale characterization enable CXL performance modeling and optimization.*

3 Overview and CXL Characterization

3.1 SupMario Overview

Figure 2 provides a high-level overview of SupMario pipeline. To address the research questions raised in §1, we need to overcome the following challenges:

- Lack of fine-grained profiling tools for **in-depth** analysis of CXL's unique performance characteristics at **request-level**, and their impact **at scale**, rather than focusing solely on high-level average latency and bandwidth to understand a limited set of workloads as in prior works [48, 49].
- Lack of **systematic** approaches to analyze CXL-induced slowdowns and identify the **root causes** of performance degradation, rather than treating CXL as a black box.
- Lack of **explainable** performance metrics to improve the observability of both 1-tier and 2-tier (with NUMA/CXL) memory systems, particularly under long memory latencies, rather than relying on heuristics.
- Lack of **deterministic** and CXL-aware data placement policies to exploit CXL performance potentials in memory interleaving and tiering setups.

SupMario introduces a suite of new benchmarking and profiling tools, analytical and modeling approaches, findings, and memory policies to bridge the gaps. For the first time, SupMario provides a detailed analysis of the *unpredictable CXL latencies and their impact on CPU efficiency*. It aims to distill key findings applicable to a wide range of workloads and unify them into a set of performance metrics and models using a simple yet accurate approach based on the novelty combination of a few CPU performance counters. The insights derived from SupMario's characterization and modeling provide deeper understanding of how CXL's long latencies affect CPU performance. Notably, we find that although SupMario performance models are specific to certain hardware configurations (*e.g.*, CPU and memory), they are **independent** of the workloads, allowing them to be applied across both offline and online scenarios. SupMario-powered memory tiering and interleaving policies not only deliver superior performance gains but also provide valuable insights for designing future CXL-aware memory systems.

3.2 Platform

We show the details of our hardware platform in Table 1.

			Local		+NUMA		Specification
CPU / CXL	**DDR** Type	**Size** GB	**Lat** ns	**BW** GB/s	**Lat** ns	**BW** GB/s	L1D-L2-L3 / CXL-dev-spec
SPR2S	16×DDR5	256	114	218	191	97	48KB-2MB-60MB
EMR2S	16×DDR5	256	111	246	193	120	48KB-2MB-160MB
SKX2S	16×DDR4	192	90	52	140	32	32KB-1MB-13.8MB
SKX8S	16×DDR4	384	81	109	411	7	32KB-1MB-38.5MB
CXL-A	2×DDR4	128	214	24	375	14	ASIC, CXL1.1, ×8
CXL-B	2×DDR5	128	271	22	473	13	ASIC, CXL1.1, ×8
CXL-C	2×DDR4	16	394	18	621	14	FPGA, CXL1.1, ×8
CXL-D	4×DDR5	768	239	52	333	14	ASIC, CXL1.1, ×16

Table 1: **Experimental platform.** *"Local" refers to the performance measured by CPUs on the same socket while "+NUMA" indicates memory access from a remote socket.*

Servers. We use two servers equipped with Intel's 4th (Sapphire Rapids, SPR) and 5th (Emerald Rapids, EMR) generation Xeon scalable server processors. The two servers are identical except for their CPUs. Each server is a dual-socket (2S) system with 16 cores per socket, running at 2.1GHz. They are equipped with 48KB L1 data cache, 2MB L2 cache, and 8 memory channels with 128 GB of DDR5-4800MHz memory. The key difference between them is the size of the L3/LLC cache: our EMR has a 160MB LLC, whereas SPR has only 60MB. As a more recent processor, EMR offers better support for CXL and delivers up to 28% better performance than SPR for certain workloads we measured (due to its much larger LLC).

We also use two Skylake servers – one with 2 sockets (SKX2S) and another with 8 sockets (SKX8S) – to extend the range of memory latencies from 140 to 410 ns using zNUMA and by lowering the CPU uncore frequency. Together, the setups provide a total of 7 latency configurations (including 4 CXL devices). We find that the performance of zNUMA and local DRAM is more stable compared to real CXL devices, making zNUMA a clean-slate environment for our characterization and modeling (further details to follow).

CXL devices. We use 4 CXL memory expanders from different vendors (denoted as **CXL-A**, **CXL-B**, **CXL-C**, **CXL-D**). Our CXL devices' average latency and bandwidth are 214-394ns and 18-52GB/s, respectively, measured by Intel Memory Latency Checker (MLC) [9]. Note that CXL-D is hosted on a remote machine while others are in our lab environment, CXL-C only supports 16GB DRAM, thus we were only able to finish a subset of 265 workloads on them.

All our CXL devices are CXL 1.1 type-3 memory expanders (supporting CXL.io and CXL.mem). These devices function as black boxes to us, as we do not have access to their internal implementation details. CXL-C is FPGA-based (lowest performance) while the rest are ASICs. CXL-D utilizes 16× PCIe 5 lanes and supports 4 DIMMs, providing the highest CXL bandwidth of 52GB/s. In contrast, the other devices use 8× lanes and 2 DIMMs, resulting in nearly half the bandwidth (18-24GB/s), as shown in Table 1. CXL-A

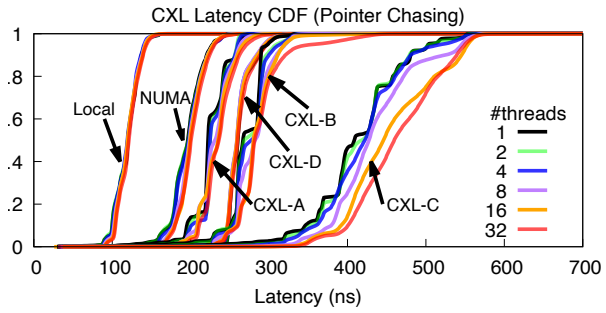

Figure 3: **CXL Latency CDF.** *Not all CXL are created equal. Unlike local/NUMA memory, CXL shows high tail latencies.*

and CXL-C use DDR4 memory, while CXL-B and CXL-D use DDR5 memory. In terms of latency, interestingly, CXL-A exhibits the lowest latency at 214ns, despite using DDR4 memory, while the DDR5-based CXL-B and CXL-D have higher latencies of 239ns and 271ns, respectively. We speculate that these differences in performance characteristics are primarily due to variations in CXL memory controller optimizations (*e.g.*, scheduling policies, row buffer management, QoS, thermal management in the controller) [36]. Accessing CXL from a remote socket (`+NUMA` column) increases the latency and decreases bandwidth. However, to our surprise, the latency increase via one NUMA hop vary more 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 (§3.4) which requires careful management.

Workloads. We use a diverse set of representative workloads for the characterization, covering cloud workloads (caching and DB such as Redis [18] and VoltDB [25], CloudSuite [1], and Phoronix [16]), graph processing (GAPBS [27], PBBS [24]), data analytics (Spark [35]), ML/AI (GPT-2 [7], MLPerf [19], Llama [11]), and high-performance computing (SPEC CPU 2017 [23], PARSEC [30]). Some workloads are latency-sensitive (*e.g.*, cloud workloads), some are bandwidth-sensitive (*e.g.*, HPC workloads), and 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.

3.3 CXL Device Characterization

We start with device-level microbenchmarks to understand CXL latency characteristics in detail. We run workloads using either local or CXL memory. Local DRAM performance is used as the baseline to calculate CXL slowdowns.

CXL latency stability and tail latencies. To understand latency variability of different CXL devices, we measure latencies for each cacheline request. As existing memory benchmarking tools do not support request-level latency reporting, we implemented a microbenchmark program (called **MIO**) that can measure cacheline-granular request latencies. MIO average latency results are validated with Intel MLC [9] reported ones to be accurate. MIO measures the average latency of each N (configurable, to amortize `rdtsc` timing overhead) pointer-chasing operations on a working set larger than LLC size. We use an in-memory buffer from an idle NUMA node to store the latency logs to avoid interference and minimize performance overhead. Figure 3 shows the CXL latency distributions of all 4 CXL devices and Local-DRAM/NUMA under 1-32 colocated pointer-chasing threads (from left to right). This setup mimics the co-location of multiple memory latency-sensitive workloads. Note that none of the CXL device bandwidth is saturated and pointer-chasing is purely latency-sensitive operation. We disabled L1/L2 prefetchers to measure device-level latencies.

We observe CXL-B and CXL-C suffers from significantly high tail latencies. Local and NUMA latencies are stable, and the difference between p99.9 and p50 latencies are only 45ns and 61ns. However, CXL latency stability largely varies across vendors. The small latency variation for local and NUMA are probably due to DRAM chip-level latency variations (*e.g.*, row buffer hit/miss, activation latencies, etc.) widely discussed in prior DRAM characterization works [31, 32, 34, 46, 55] (also in §8). Local DRAM latency variation is much smaller than that of CXL. For example, CXL-D can deliver the best latency stability, its difference between p99.9 and p50 is 75ns (only 30ns and 14ns more than Local and NUMA). However, for CXL-B and CXL-C, it can reach $\sim$160ns, which is 50% higher than the median latency. When looking at higher percentiles at p99.99 and p99.999, CXL device latencies will be above 700ns for CXL-A and CXL-D and $>1\mu$s for CXL-B and CXL-C.

Similarly, when one pointer-chasing thread is co-located with multiple bandwidth-bound read/write threads (results not shown), we observe even worse tail latency trends on CXL compared to Local/NUMA. When turning on CPU prefetchers, we see effective improvement of the average latency but tail latencies persist for CXL.

We speculate that high CXL tail latencies are caused by the CXL controller sub-optimal optimizations, for example, inefficiencies in thermal management or memory request scheduling could lead to long queueing delays. Unfortunately, there are no available tools to investigate the exact cause of CXL tail latencies. A potential future white-box approach could involve breaking down the latency of each memory request and accounting for the latency across different components, such as the CXL link, CXL controller, and DRAM chips. This would be feasible if CXL controller exposes detailed performance counters, for example, through the upcoming CXL performance monitoring unit (CPMU) defined in CXL 3.0 specification [3], similar to the CPU PMU. As a first step, we aim to demonstrate and quantify the impact of CXL tail latency to raise awareness in the systems community. To summarize,

Finding #1: Not all CXL devices are created equal, each car-

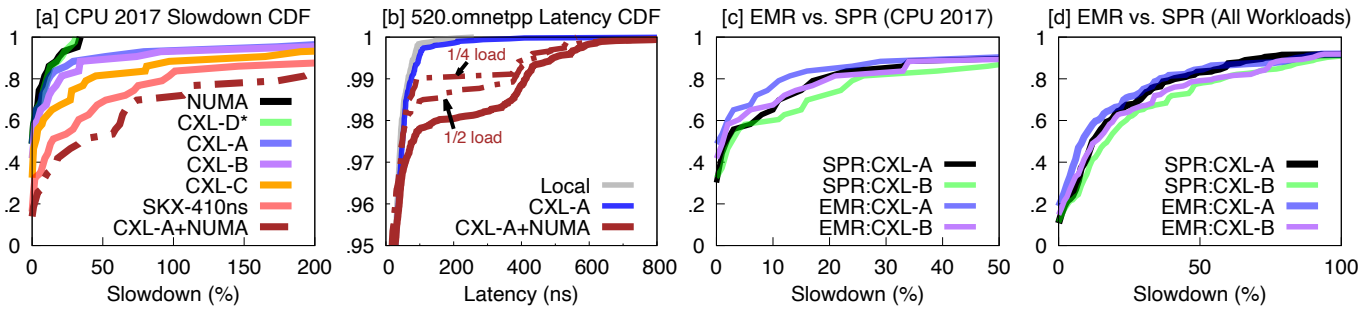

Figure 4: **CDFs of workload slowdowns under various CXL.** *(a) the CDFs of SPEC workloads on all our platforms; (b) tail latency is the cause of significant workload slowdown under CXL+NUMA for a latency-insensitive workload; (c) SPR vs. EMR SPEC results under CXL-A and CXL-B; (d) is similar to (c) but for all 265 workloads.*

rying very unique performance characteristics. More importantly, CXL devices exhibit unstable and higher tail latency compared to regular socket-local or NUMA memory. High access parallelism and high memory pressure (*e.g.*, bandwidth) can exacerbate CXL tail latencies. Further, concurrent reads and writes exert differing impacts on memory latency for CXL devices, especially regarding tail latencies. While CPU hardware prefetchers can improve average memory access latencies, they fail to mitigate tail latencies. CXL tail latencies negatively impact application performance.

Implication #1: From both software and hardware design perspectives, there is a need to address CXL tail latencies. Future CPUs need be improved (*e.g.*, via smarter CXL-aware prefetching policies) to better manage CXL's long and unpredictable latencies effectively. Additionally, (some) CXL controllers need further optimizations to achieve latency predictability, rather than solely focusing on average latency and bandwidth.

Recommendation #1: Tail latency should be used as a key metric for evaluating CXL devices, as predictable latency is crucial for meeting user service level objectives (SLOs) in cloud environments.

3.4 Workload Characterization

To fairly compare results from different CXL devices, we first analyze common workloads that we complete on all platforms followed by more workloads analysis (265) on zNUMA, CXL-A and CXL-B.

Figure 4a shows the CXL slowdown CDF of 43 workloads from SPEC CPU 2017 across 4 CXL devices on EMR and 3 zNUMA latency configurations. The left-most black line is NUMA performance with up to 34% slowdowns from two bandwidth-intensive workloads (`619.lbm` and `649.fotonik3d`). Almost half of the workloads do not experience slowdowns at all due to the large cache in EMR CPU (160MB LLC). In total, 32 workloads experience less than 5% slowdowns and 3 more workloads below 10%. Among the four CXL devices, CXL-D (green line) performs on-par with zNUMA because its high bandwidth prevents any workloads from being bandwidth-bound. There are four bandwith-bound workloads requiring over 24GB/s – `603.bwaves`, `619.lbm`, `649.fotonik3d`, `654.roms` – whose bandwidth needs exceed the capacity of CXL-{A, B, C}. As a result, these workloads experience significant slowdowns (over 50%) compared to zNUMA/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. For the remaining workloads which do not saturate CXL bandwidth, we observe the performance worsens with increasing CXL latency. For example, `602.gcc` slowdown goes up from 12% up to 13%, 21%, and 38% for CXL-A, CXL-B, and CXL-C, respectively. Other workloads might experience more significant performance impact under increased latency, *e.g.*, `503.bwaves_r` slowdown jumps from 11% to 16% (CXL-A), 33% (CXL-B), and 81% (CXL-C).

CXL-C is the least performant in the four CXL devices in terms of average latency, bandwidth, and latency stability due to the FPGA-based CXL controller implementation. It shows significantly worse slowdown results compared to CXL-A and CXL-B. For example, `649.fotonik3d` even sees a 5.3× slowdown, showing a combined impact from long (unpredictable) latency and low bandwidth.

(Suspicious) CXL+NUMA performance. We planned to use CXL+NUMA setup to simulate CXL memory access latency setups in the range of 400-700ns. However, we find workload performance under CXL+NUMA is significantly worse even than that of 2-hop NUMA whose latency and bandwidth are both worse, indicating issues when CXL and NUMA are used together. In CXL+NUMA, memory requests need to go through cross-socket interconnect (*e.g.*, UPI) first before reaching the CXL device. CXL+NUMA results are shown in the "CXL-A+NUMA" dotted brown line in Figure 4a. Surprisingly, while CXL+NUMA latency is lower than SKX-zNUMA (375ns vs. 411ns) and bandwidth is higher (14GB/s vs. 7GB/s), CXL+NUMA performance is much worse than CXL-C, which does not seem to make sense. Similarly, this is true for CXL+NUMA vs. CXL-C where CXL+NUMA latency is lower (375ns vs.

394ns). Note CXL+NUMA bandwidth is indeed lower than CXL-C (14GB/s vs. 18GB/s), but when filtering out workloads needing more than 10GB/s bandwidth, CXL+NUMA slowdowns are still much worse than CXL-C. For example, `520.omnetpp` sees <5% slowdowns under all CXL devices, but experiences an astonishingly high slowdown of 2.9× under CXL+NUMA. Upon further analysis, we found this workload consumes <1GB/s bandwidth (read+write), and is neither latency-sensitive or bandwidth-sensitive. 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. In Figure 4b, we show the CDF of sampled memory latencies for the workload. The plot shows little difference between Local and CXL-A (gray and blue lines), which explains the small slowdown under CXL-A. However, CXL+NUMA (brown line) exhibits a long tail latency starting around p98 up to 800ns. As we reduce the load of the workload (by reducing the number of simulated LANs on backbone switches) to 1/2 and 1/4, we observe consistently improved tail latencies (two dotted brown lines). Correspondingly, the slowdown on CXL+NUMA also significantly decreases from ∼290% down to ∼65% and 58%. We believe this serves as direct evidence that tail latencies are the root cause of the performance slowdowns. Similarly, 10 other workloads do not experience noticeable slowdowns under CXL but 33%-283% under CXL+NUMA. These findings are consistent for both SPR and EMR, and persist regardless of CXL device used.

SPR vs. EMR. Figure 4c compares the slowdowns for SPEC workloads under SPR and EMR. Compared to SPR, EMR features a larger LLC size and microarchitecture optimizations for CXL, which might lead one to expect improved performance. However, Figure 4c shows that the CXL slowdowns with EMR are not significantly reduced despite the increased LLC size, indicating that larger caches have limited effectiveness in mitigating the impact of long CXL access latencies. Although EMR shows slightly less slowdowns than SPR on both CXL-A and CXL-B, the CXL-induced slowdowns largely persist. This indicates that existing caches and/or prefetchers are not effective at hiding long memory latencies. These findings suggest that simply increasing CPU cache size is insufficient for optimizing CXL. Future CPU designs will need to incorporate further optimizations to better mitigate the impact of CXL's long latencies.

All workloads. Figure 4d presents the slowdown CDF for 265 workloads on both EMR/SPR and CXL-A/CXL-B. Compared to the CPU 2017 results in Figure 4c, the slowdowns are more prounced as workloads from other benchmarking suites, such as graph and ML/AI, tend to be memory-intensive, leading to greater performance degradations. However, the overall performance patterns remain consistent, *e.g.*, EMR outperforms SPR (albeit by a small margin) on both CXL devices. On EMR, more than 15% of the workloads experience over 50% degradation on CXL-A, while this percentage increases to 20% for CXL-B due to its higher latency (and/or less predictable latency). For SPR, 16% and 22% of workloads exhibit over 50% performance degradation on CXL-A and CXL-B, respectively. The slowdown CDFs also reveal a clear "tail," with 5% of the workloads suffering from slowdowns of 2.3-6.3×, primarily due to being bandwidth-bound.

In summary, the key takeaways from the workload-level charcterizations are as follows:

Finding #2:

- Workload performance deterioates superlinearly with increasing CXL latency; more importantly, the relative slowdowns exceed the rate of the latency increases).
- Longer CXL latencies correspond to worse bandwidth (CXL A→B→C), which has a more pronounced impact on bandwidth-bound workloads than purely latency-sensitive workloads due to the combined effects of increased latency and limited bandwith.
- CXL devices with worse tail latencies (*e.g.*, CXL-B and CXL-C) experience more significant slowdowns across all evaluated 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 real-world applications in pooling scenarios.

Implication #2: As future CXL devices are expected to significantly increase bandwidth (CXL-D is a good example, and bandwidth can also be easily enhanced through hardware interleaving across multiple CXL devices) and moderately reduce latency, we anticipate that future CXL workload slowdowns will be smaller than those shown in Figure 4a. Higher CXL bandwidth will benefit bandwidth-bound workloads, potentially alleviating the 2-6× slowdowns observed in Figure 4a due to the low bandwidth of individual CXL devices. Reductions in latency will improve the performance of latency-sensitive workloads, such as cloud applications, bringing it closer to NUMA performance.

Recommendation #2: CXL latency is more critical to performance when bandwidth is no longer a bottleneck (see Figure 4a) and deseves more attention in future CPU/CXL designs as well as software optimizations. However, for bandwidth-bound workloads to effectively utilize the combined bandwidth of local and CXL memory, improved software approaches are still needed.

4 Performance Modeling

4.1 Slowdown Root-Cause Analysis

Our goal is to break down workload slowdowns into contributions from the CPU cache hierarchy and CXL memory.

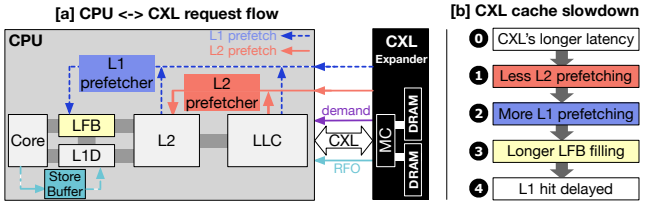

Figure 5: **CXL slowdown breakdown.** *Figure (a) shows various components where CXL introduces overheads; Figure (b) details the flow of CXL-induced cache slowdowns.*

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 [40], we will *measure* CXL's impact on prefetcher performance and *disclose* why it happens.

To achieve this, we need an approach to *capture* the events in the CPU pipeline that lead to performance slowdowns under CXL and correlate them accurately back to workload-level slowdowns. The extensive microarchitecture-level information offered by CPU PMU counters provides valuable insights into the efficiency of the CPU pipeline. While workload slowdowns can be directly measured using application-level metrics, identifying the underlying PMU events/metrics that can correlate to the slowdowns is often challenging. It is even more challenging to establish a precise correlation between workload performance and architecture-level performance metrics. The Intel TMA method [54] is a popular approach for top-down performance analysis, but it is insufficient for our objectives.

1. TMA identifies dominant performance bottlenecks in an application by analyzing execution inefficiencies within the CPU pipeline for a fixed setup using either local DRAM or CXL memory. However, it does not provide a differential analysis to interpret pipeline differences resulting from varying backend memory.
2. Although a differential analysis can be done manually, there is *no method to precisely correlate microarchitecture level metrics with workload slowdowns*. The TMA metrics are designed to capture the performance or contention of specific hardware components rather than overall workload behavior.

For these reasons, we begin by examining components of the CPU pipeline involved in instruction execution and analyzing the **changes** induced by CXL on those components during memory request processing. 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. Figure 5a highlights the key components as observation points for memory request processing during

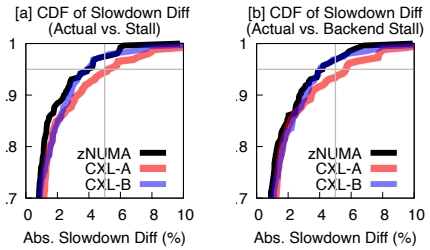

Figure 6: **CDFs of slowdown differences using stalls and backend stalls.** *The X-axis represents the absolute difference between estimated slowdowns using stalls or backend stalls and the actual measured slowdowns for each workload.*

CPU and CXL interactions. Through detailed offline analysis, we make a few key observations that lead to an accurate slowdown breakdown method which we describe below.

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 $S = \frac{c'-c}{c} = \frac{\Delta_c}{c}$.

Finding #3: The variance in cycle counts between CXL and local DRAM primarily stems from *stall cycles* difference, which further mainly arises from the CPU pipeline *backend*.

As such, CXL slowdowns can be estimated as:

$$S = \frac{\Delta_c}{c} \approx \frac{\Delta_{stall}}{c} \approx \frac{\Delta_{backend-stall}}{c} \quad (1)$$

CPU backend refers to memory-subsystem. Purely CPU-bound workloads are not sensitive to CXL latency due to few CXL accesses, thus experiencing minimal slowdowns.

Accuracy. To validate the finding, we measure (backend) stall cycles for each workload and use them to estimate the workload slowdowns according to (1). We compare them with the actually observed workload slowdowns using application-level metrics (*e.g.*, time, throughput). Figure 6 presents the CDF plots of the absolute difference between the actual slowdown and the (backend) stall based slowdown estimations, which indicates the inaccuracies. We show the results for zNUMA, CXL-A, and CXL-B. We observe very low inaccuracies – within 5% for over 95% of workloads (the intersection of two gray lines). Therefore, CXL-induced (backend) stall cycle difference can effectively represent the slowdown.

Implication #3: Workload slowdowns on CXL are primarily due to the additional backend stalls, which are caused by memory subsystem inefficiencies.

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. Stalled cycles can occur due to stalls in either the frontend, the backend, or both. However, 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. Therefore, it is primarily stalls in the memory subsystem (*i.e.*, the CPU backend) that are impacted by CXL. As a result, stalled cycles in the memory subsystem can serve as a suitable approximation for slowdown caused by CXL.

Breaking down the slowdown. Figure 5a highlights the simplified CPU backend components where the majority of these stall cycles occur, including the *store buffer* for serving writes, *L1–LLC*, and *CXL* for serving reads. By observing the number of stall cycles on each component, we can further understand how (much) each of these backend components contribute to workload slowdowns.

On Intel platforms, the stalls on the store buffer, L1, L2, LLC, and (CXL) DRAM represent exclusive events which sum up to the total backend stall cycles (see Figure 4 in [54]). Let s be the number of stall cycles, according to TMA approach, we have:

$$s_{Local} = s_{store} + s_{L1} + s_{L2} + s_{L3} + s_{DRAM} \quad (2)$$

$$s_{CXL} = s'_{store} + s'_{L1} + s'_{L2} + s'_{L3} + s'_{DRAM} \quad (3)$$

In the above formula, s_{L1} and s'_{L1} denote the number of stall cycles on local and CXL memory, respectively, due to L1 cache accesses. Other terms follow a similar definition. When looking at the difference between the two, we get:

$$\Delta_{stall} = s_{CXL} - s_{Local} = \Delta s_{store} + \Delta s_{L1} + \Delta s_{L2} + \Delta s_{L3} + \Delta s_{DRAM} \quad (4)$$

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 can be represented as the combined slowdowns from the five sources as follows:

$$S \approx S_{store} + S_{L1} + S_{L2} + S_{L3} + S_{DRAM} \quad (5)$$

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 stalled cycles of LLC misses, 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. We also identify memory level parallelism (MLP) as another key metric for analyzing slowdowns. Later, we will show how it enhances slowdown prediction in §5.3.

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.

4.2 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). Similarly, they can be measured using the corresponding stall cycles counters. Below we describe our findings to reason about cache slowdowns on CXL through offline analysis.

Finding #4:

1. Cache slowdown under CXL is due to reduced prefetch efficiency. To validate this, we disable all the hardware prefetchers (L1 and L2, LLC-prefetcher is disabled by default) and measure workload slowdowns. With prefetchers off, we found virtually no stall cycles on cache ($S_{L1} = S_{L2} = S_{L3} = 0$).
2. Through our extensive offline analysis, we find CXL's relatively longer latency causes L2-prefetcher inefficiency (less useful data in L2 cache), thus causing L1-prefetcher to fetch more data from LLC/CXL. As a result, L1 demand reads are affected negatively (more stalls in L1), thus causing cache-slowdown.
3. Upon further analysis, we find cache slowdown is mainly reflected as the increase of hits on line fill buffer (LFB), a per-core small buffer with 10-20 entries that connects L1 and L2 caches. Due to the reduced L2 prefetcher efficiency, L1 prefetcher fetches more data from LFB, causing higher LFB hits.

To summarize, as shown in Figure 5b, CXL initially leads to reduced efficiency of L2 prefetchers. With less useful data in L2 cache, L1 prefetchers are compelled to fetch more data from LLC or (CXL) DRAM due to L2 misses. Moreover, CXL affects L1 prefetch efficiency as well. Data fetched by L1 prefetchers must be temporarily stored in LFB before reaching L1 cache, and this would cause more requests to be served by (slower) LFB hits instead of direct L1 hits, causing L1 slowdowns.

Reasoning of reduced L2-prefetch efficiency under CXL. Through offline analysis of cache-related PMU counters for local-DRAM and CXL, we find reduced number of L2 prefetch requests that misses L3/LLC (L2-prefetch-L3-miss) on CXL. Meanwhile, L1 prefetch that misses L3/LLC (L1-prefetch-L3-miss) increases. The increase is almost the same as the decrease of L2-prefetch-L3-miss, as shown in Figure 7a, while L2-prefetch-L3-hit does not change. The decrease of L2-prefetch-L3-miss on CXL setup indicates the L2-prefetcher fails to fetch as much data as on local-DRAM-setup from CXL, thus reducing L2-prefetcher

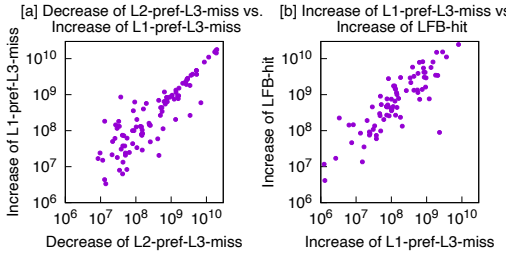

Figure 7: **Correlations of LFB-hit, L1-prefetch-L3-miss and L2-prefetch-L3-miss.** *(a) shows strong linear correlations of L2 prefetches that miss L3 and increase of L1 prefetches that miss L3; (b) shows a similar trend for increase of L1 prefetches that miss L3 and increase of LFB hits.*

efficiency. As a result, the L1-prefetcher can't find data from L2 cache that should be fetched by L2-prefetcher and it has to fetch more data from CXL, which explains the increase in L1-prefetch-L3-miss. Figure 7a shows that the decrease of L2-prefetch-L3-miss has a strong positive relationship with the increase of L1-prefetch-L3-miss (almost $y = x$), with a Pearson coefficient of 0.99.

Cache slowdown can be observed via LFB-hit increases. LFB connects L1 and L2 caches. The data of all read requests must be placed in the LFB before reaching L1 cache from L2 or lower levels, as in Figure 5a. Due to its limited size, LFB can become a bottleneck for data flowing to L1 cache. For example, Figure 7b shows that the increase in L1-stalled-cycles correlate with high pressure on LFB (more LFB hits), caused by L1-prefetch-L3-miss increasing. Particularly, the increase in LFB hits (difference between CXL and Local-DRAM) is (almost) *linearly* correlated with the increase in L1-prefetch-L3-miss. It means that more data is fetched from CXL to L1 cache by the L1-prefetcher, which becomes LFB hits.

Similarly, the increase in LFB hits is positively correlated with the decrease in (demand read) L1 cache hits. The reason is that the data fetched by L1-prefetcher first goes to LFB, but has not yet been transferred to L1 cache, due to the longer memory latency of CXL. The data required by load instruction is fed by LFB but not L1 cache, resulting in L1 hit becoming delayed hit on LFB.

In summary, if a workload heavily relies on data from L1 prefetch (*e.g.*, sequential, stride, or streaming access), and this data primarily originates from DRAM, with subsequent data often in the same cacheline, then the stall cycles of L1 demand misses may worsen. Consequently, such workloads are prone to experiencing high L2 cache slowdown under CXL. We also observed that on SPR/EMR, cache slowdown predominantly arises from LLC rather than L2 (SKX+zNUMA), validated similarly.

Next, we will apply this approach to various workloads. Our aims are twofold: validate the plausibility of our assumptions; and illustrate how the breakdown method can reveal interesting insights overlooked in prior research.

4.3 Workload Slowdown Diversity

Figure 8 depicts the overall and breakdown of CXL slowdowns for each workload under zNUMA, CXL-A and CXL-B. "Other" indicates the slowdown contribution which is not captured via our analysis. The breakdown allows us to further analyze various causes of CXL slowdowns. Below we summarize some findings.

For different workloads, the contribution of slowdown from various sources varies. Taking SPEC workloads such as `519.lbm`, as an example, the majority of the slowdown originates from stalls in the CPU's store buffer. This indicates a high volume of RFOs and insufficient entries in the CPU's store buffer. These observations are further supported by observations such as high UPI non-data traffic and high write bandwidth. However, in workloads like `649.fotonik3d`, a significant portion of the slowdown arises from the cache.

For GAPBS workloads, the primary source of slowdown is from DRAM (stalls in LLC miss demand reads). Only a few, such as `bc-urand`, `sssp-web`, and `bfs-urand`, encounter slowdown from the cache. Many of the Llama workloads experience L3/LLC slowdowns. Cloud workloads such as Redis and VoltDB, mainly suffer from DRAM slowdowns. Similarly, DRAM slowdowns take up 90% of the overall slowdowns for ML workloads like DLRM and GPT-2.

Figure 9 shows the CDFs of slowdowns caused by various components. Briefly, at least 15% workloads experience at least 5% cache slowdown on CXL, indicating the degraded prefetch efficiency under CXL. Meanwhile, at least 40% workloads experience with at least 5% DRAM slowdown. Interestingly, L2 cache slowdown prevails as the dominant factor across all examined workloads in the breakdown analysis (on SKX-zNUMA). Notably, deteriorated memory latency and decreased memory bandwidth contribute to an upsurge in stalled cycles in the L2 cache. Additionally, the stalled cycles in L1 and L3 remain relatively unaffected.

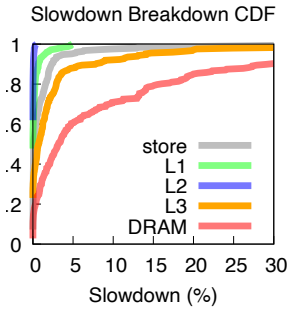

Figure 9: **CDFs of slowdown breakdown.**

Certain workloads, such as `627.cam4`, `607.cactusBSSN`, and `602.gcc`, demonstrate similar CXL performance slowdowns. However, the reasons behind the performance slowdowns vary significantly among them. In `602.gcc`, half of the slowdown stems from LLC misses, while the other half arises from cache. Conversely, almost all slowdown in `607.cactusBSSN` results from LLC misses, while for `627.cam4`, reads caused by stores (RFOs) dominate the performance slowdown. This underscores one of the advantages

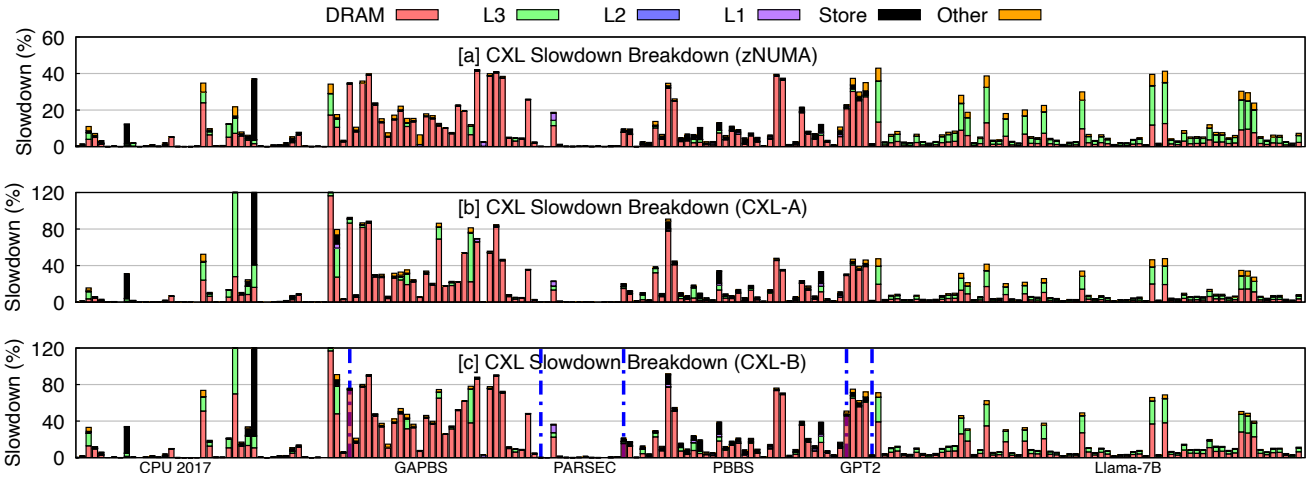

Figure 8: **CXL slowdown breakdown.** *This figure shows the CXL slowdown breakdown on zNUMA, CXL-A, and CXL-B.*

of the breakdown method, as it highlights that although the performance slowdowns may appear similar, the underlying causes can be vastly different.

To summarize, our approach could capture, explain and breakdown CXL slowdowns based on the CPU stall cycles approach. Later, we will further enhance our approach for CXL performance prediction to show its efficacy.

5 CXL Slowdown Prediction

The capability to predict system performance is appealing due to its wide range of applications. Our previous root-cause analysis of CXL slowdowns has helped identify various sources of slowdown, which, when combined, can facilitate reasoning about measured CXL performance. In this section, our objective is to transition and solidify our breakdown analysis into formal prediction models. In particular, when the model is used together with an offline workload run on local DRAM, it can accurately predict the amount of slowdowns when the workload runs on CXL. Later, we will also show the prediction model can be used in an online fashion for performance optimations.

5.1 Strawman

We initially explore simple correlations between commonly used performance metrics such as LLC miss rate, represented as misses-per-kilo-instructions, (MPKI, Figure 10a), read memory bandwidth (Figure 10b), and TMA DRAM-bound metric (Figure 10c), as they are used in many prior works [41, 48]. However, none of these metrics prove reliable as performance predictors. For example, despite a positive relationship between read bandwidth and the overall slowdown, read bandwidth falls short as a reliable predictor. Workloads with similar bandwidth often experience varying CXL slowdowns, *e.g.*, 5-50% under 10-20GB/s. We attribute this to the limitations of the aforementioned metrics in capturing CXL slowdowns across diverse sources. This prompts us to develop separate prediction models for cache, DRAM, and store-induced slowdowns. These efforts result in several simple models, which can be combined to accurately predict overall CXL slowdowns, relying solely on 12 counters on SPR/EMR (11 on SKX).

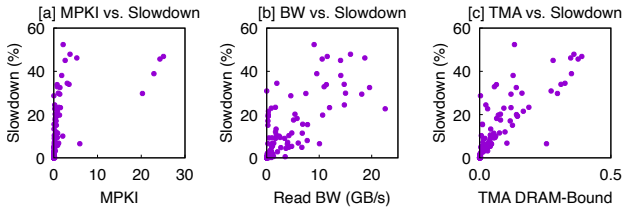

Figure 10: **Strawman prediction.** *Metrics like MPKI, BW, and TMA DRAM-Bound are not reliable CXL slowdown predictors.*

5.2 Latency and Bandwidth Sensitivity

Workload performance is influenced by both memory latency and bandwidth. Bandwidth-sensitive workloads can benefit from increased memory bandwidth through technologies like CXL, while latency-sensitive workloads are better managed with tiering strategies to mitigate latency impacts. Therefore, accurately determining a workload's sensitivity to bandwidth or latency is crucial.

We propose using a CPU offcore latency-based model for this purpose. Our benchmarking results indicate that under bandwidth contention, queueing delays contribute to end-to-end request latencies. Offcore latency reflects both memory latency and bandwidth-induced overhead. A simple heuristic is to set an offcore latency threshold. If latency exceeds this threshold, it indicates bandwidth limitation; otherwise, it is latency-bound. Additionally, the offcore latency threshold can be easily profiled using pointer-chasing style workload, as in SupMario tail latency analysis.

We used this approach to filter out bandwidth-bound

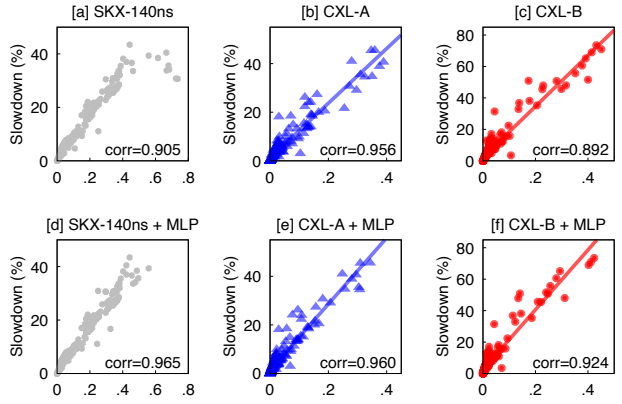

Figure 11: **DRAM slowdown model.** *X-axis is our predictor (discussed later) and Y-axis is measured DRAM slowdown. 182 latency-sensitive workloads are shown. (a)-(c) show the basic DRAM model by using "l3-stalls/cycle" as the predictor for SKX-zNUMA, CXL-A, and CXL-B respectively; (d)-(f) represent the enhanced DRAM model incorporating memory level parallelism (MLP) impact, improving the model accuracy.*

workloads on CXL, which experience much higher slowdowns (*i.e.*, the tail in Figure 4d) where slowdowns can be up to 6×.

5.3 DRAM (Load) Slowdown Model

There are two insights in our DRAM slowdown prediction. The first is the overall ratio of stalled cycled on LLC (*i.e.*, "P_4 / P_1") as a base predictor can already positively correlate with DRAM-sourced slowdown. We started this analysis on SKX2S zNUMA. In Figure 11a, we correlate the based predictor observed when the workloads run under local DRAM (90ns) with the DRAM-slowdown in zNUMA (140ns). Notably, the predictor does a great job for most workloads showing a strong linear relationship, with a few outliers on the top right, indicating the predictor is mistakenly overpredicting the slowdowns.

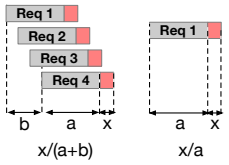

Figure 12: **CXL MLP**

Second, we argue that not taking high **memory-level parallelism (MLP)**, more precisely, overlapping effect, into account is the cause of the above outliers. As shown in Figure 12, CXL has the same impact on each single data request. For each data request, the latency will be increased similarly, *e.g.*, x. However, under high MLP, the overlapping effect lowers the CXL impact on the slowdown (DRAM load), as in Figure 12 left, reducing the latency from x/a to $x/(a+b)$ (b indicates the stalled cycles from previous demand requests caused by overlapping). A large amount of demand reads could cause considerable LLC miss stalls, but the increase of stalled cycles of previous demand read misses could be overlapped by the last several demand read misses. The increased LLC miss stalls of part of the data requests impacted from low memory latency efficiency could be overlapped by the other demand reads. In contrast, if the demand reads are spaced out, more demand reads could be affected by memory latency and further influence the overall increase of LLC stalls caused by remote (CXL) memory. Therefore, we assume that the degree of overlapping would decrease the CXL impact on the (DRAM) slowdown.

Unfortunately, this effect cannot be directly measured. Instead, we choose to approximate it using the amortized offcore demand read latency. By incorporating MLP into the model, Figure 11d shows a much stronger linear relationship (Pearson coefficient goes up from 0.905 to 0.965).

Accuracy on SPR/EMR with real CXL. Figure 11b-c&e-f show the DRAM slowdown models for CXL-A and CXL-B. Similar to zNUMA, it could predict the DRAM slowdown reliably. Applying MLP impact to the model still helps improve model accuracy on SPR/EMR, but less so compared to SKX. We speculate this is because latest EMR CPUs with large LLC cache experiences less MLP, thus less outliers caused by it. Table 2 shows the store slowdowns of **92.0%**, **94.0%** and **78.7%** workloads can be predicted within **5%** deviation on zNUMA, CXL-A and CXL-B, respectively.

	zNUMA	CXL-A	CXL-B
Pearson Correlation Coefficient	0.965	0.960	0.924
Absolute Error within 5%	92.0%	94.0%	78.7%
Absolute Error within 10%	99.1%	98.3%	89.9%

Table 2: **DRAM slowdown prediction accuracy.** *We can achieve 78.7%–94% accuracy under 5% misprediction target while the accuracy goes up to 89.9%–99.1% under 10% misprediction.*

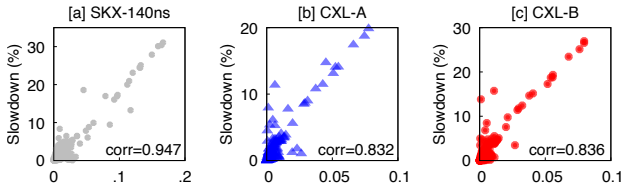

Figure 13: **Cache slowdown model.** *X-axis is our predictor, Y-axis is actual cache slowdown. (a)-(c) for SKX zNUMA, CXL-A and CXL-B for all workloads.*

5.4 Cache (Load) Slowdown Model

Cache introduced slowdowns are hard to directly measure and quantify. We develop a metric to predict cache slowdowns based on our root cause analysis. Workloads spending more stalled cycles on L2 cache, accessing increased data on LFB, allocated by L1 prefetching requests missing on L3, and primarily prefetched from DRAM by L1 prefetchers, may encounter elevated cache slowdowns. Leveraging pertinent performance counters, our predictor aims to effectively

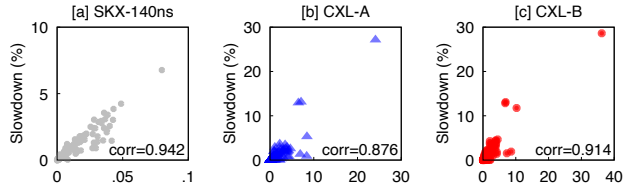

Figure 14: **Store slowdown model.** *Similar to cache slowdown model in Figure 13.*

	zNUMA	CXL-A	CXL-B
Pearson Correlation Coefficient	0.947	0.832	0.836
Absolute Error within 5%	95.8%	96.5%	93.6%
Absolute Error within 10%	99.2%	98.8%	98.2%

Table 3: **Prediction accuracy of cache slowdown.**

capture the contention indicative of cache slowdowns.

Intel offers counters for helping derive performance predictor (M_{cache}) for cache slowdown (S_{cache}).

Table 3 shows the store slowdowns of **94.7%**, **83.2%** and **83.6%** workloads can be predicted within **5%** deviation on zNUMA, CXL-A and CXL-B, respectively.

5.5 Store Slowdown Model

We find that bound-on-store counter is positively related to store slowdown. It means that on remote memory (CXL), it is increased by the same factor for most of the workloads.

Table 4 shows the store slowdowns of **97.8%**, **87.6%** and **91.4%** workloads can be predicted correctly within **2%** deviation on zNUMA, CXL-A and CXL-B, respectively.

5.6 Put It All Together

Each component contributing to the breakdown of slowdown can be individually predicted by our introduced model. The overall slowdown is determined by summing the slowdown from DRAM, cache, and store. Given that most real-world servers share similar architectural organizations, such as multiple cache levels, a store buffer, LFB, SQ, and L1/L2 prefetching, we believe this methodology can be universally applied across different server models to analyze and predict performance slowdowns caused by sub-µs memory latencies.

The limited availability of CPU counters impacts the accuracy of performance modeling under CXL. Nevertheless, we demonstrate that by meticulously integrating multiple counters in a novel manner, we can effectively capture system performance and use it for reliable performance prediction.

The overall slowdown model (S) is described below. P_1–P_{12} are the CPU counters needed for the DRAM (M_{DRAM}), cache (M_{cache}), and store (M_{store}) performance predictors. k_1, k_2, k_3, k_4 are constants.

$$S = k_1 \times M_{DRAM} + k_2 \times M_{cache} + k_3 \times M_{store} + k_4$$
$$M_{DRAM} = P_4/P_1 \times 1/(p \times 1/(P_{12}/P_{11}) + q)$$

	zNUMA	CXL-A	CXL-B
Pearson Correlation Coefficient	0.942	0.876	0.914
Absolute Error within 2%	97.8%	93.7%	95.6%
Absolute Error within 5%	99.1%	97.5%	98.1%

Table 4: **Prediction accuracy of store slowdown.**

$$M_{cache} = (P_3 - P_4) \,/\, P_1 \times P_6 / (P_5 + P_6) \times P_{13} / P_{14} \times P_{15}/(P_{15}+P_{16})$$
$$M_{store} = P_7/P_1$$

Mispredictions. Mispredictions may arise partly due to the absence of certain performance counters provided by Intel. First, measuring the proportion of L1 prefetching data requests within LFB hits is impractical. Second, gauging L1 prefetching hits on L2 cache, even with the total number of prefetching data requests from LFB hits known, remains unfeasible. Therefore, we solely employ the L1 prefetching L3 miss ratio to represent the ratio of data prefetched directly from DRAM by L1 over the total number of data prefetched by L1 on LFB. This explains the outliers in SKX. SPR/EMR has the simialr issues on SQ. Moreover, SPR/EMR does not support measuring L1/L2 prefetching offcore hit for each process. This limitation explains the slightly worse predictions on SPR/EMR.

Deployment. To derive the DRAM slowdown model for a server and CXL configuration, users do not need to go through the extensive characterization process as we do because our models are robust and independent of workloads. Thus, one could use microbenchmarks (*e.g.*, pointer chasing) to derive the parameters of the linear model. Below we provide a high-level overview of the process. There are several constants (k_1-k_4, p, q) in our prediction model equation. They can be obtained by a set of microbenchmarks with distinct memory access patterns.

We rely on three types of microbenchmarking workloads to derive the model parameters. The first one is a random pointer-chasing workloads, which imposes zero cache and store slowdowns. It can be seen as having pure DRAM slowdown on CXL. After running it on both local and CXL memory, we can get the overall slowdown (S) and calculate the DRAM metric (M_{DRAM}). Due to S_{store} and S_{cache} being 0, k_1 will be S/M_{DRAM}.

Our next microbenchmark is a store-bound workload, *e.g.*, one with many `malloc()`. It does not experience cache slowdown, because its accesses do not rely on data from prefetchers. In this case, $S = (k_1 * M_{DRAM}) + (k_3 * M_{store})$. After the microbenchmark running on both local and remote, S is the overall slowdown. M_{store} and M_{DRAM} can be obtained from the local run. Then k_3 could be calculated .

To reveal cache slowdown, the third microbenchmark conducts linked-list traversal, which requires data fetched by prefetchers. After running it on both local and remote, S can be obtained by the running time. And $S = (k_1 * M_{DRAM}) + (k_3 * M_{store}) + (k_2 * M_{cache})$. Indeed, $(k_3 * M_{store})$ should be

small because it has few data allocations or writes. M_{DRAM} can be calculated with CPU counters measured on local. Given by S, k_1, k_3, M_{store} and M_{DRAM}, k_2 can be obtained. Finally, to improve accuracy, one could consider mixing up these types of memory access and also applying linear fitting.

6 System Optimization

In this section, we show SupMario root-cause analysis approach and slowdown prediction models can aid inefficiency detection and performance optimizations under complex memory policies, such as interleaving and tiering.

6.1 Interleaving Characterization

NUMA interleaving can potentially speedup bandwidth-bound workloads by leveraging the additional CXL bandwidth alongside system memory. Recently Linux kernel introduced weighted interleaving policy [26] which allows more flexible page interleaving across two or more NUMA nodes. Traditionally, Linux defaults to `1:1` page interleaving (*i.e.*, `MPOL_INTERLEAVED`) where page allocations are done in round-robin fashion across multiple NUMA nodes. Weighted interleaving define a general `M:N` interleaving ratio so that one could use the bias to match the bandwith characteristics. For instance, in a two-node system, weighted interleaving `M:N` means that the first `M` pages are allocated from one node, followed by the next `N` pages from the other node, alternating between the two in this pattern. However, it is not intuitive to decide the best interleaving ratio to extract the best potential performance for certain workloads.

For bandwidth-bound workloads, it is alluring to exploit both system and CXL bandwidth to improve performance. Suppose system memory bandwidth is `M` and CXL memory bandwidth is `N`, simply using an interleaving ratio of `M:N` do not always lead to the best performance. A naive approach is to randomly try out `M × N` interleaving ratios which can be extremely time-consuming for long-running workloads. In a recent work, Caption [48] proposed heuristics to converge over the best interleaving ratio setup but still requires a few runs and could potentially lead to suboptimal results.

We show that by adopting a performance "slowdown" prediction model for interleaving similar to §5, we can predict the best page interleaving ratio for best performance, thus "best-shot interleaving."

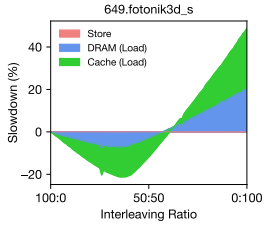

Figure 15: **Weighted interleaving performance.**

Offline Analysis. We conducted an extensive offline analysis across 100 different local/CXL interleaving ratios (100:0, 99:1 to 0:100), for over 100 workloads. Figure 15 illustrates one such example for workload `649.fotonik`. The study yields following key insights used to develop the best-shot interleaving model.

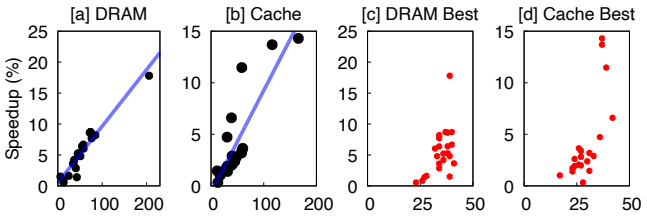

Figure 16: **Best-shot interleaving for SKX.** *In (a) and (b), X-axis is our predictor (R), and Y-axis is the (predicted) speedup from DRAM and cache respectively. The black circles are offline optimal interleaving results. In (c) and (d), X-axis is the best interleaving ratio assigned to zNUMA and Y is the actual interleaving performance speedup sourced from DRAM and cache. The best interleaving ratio on SKX ranges from 34%-40%.*

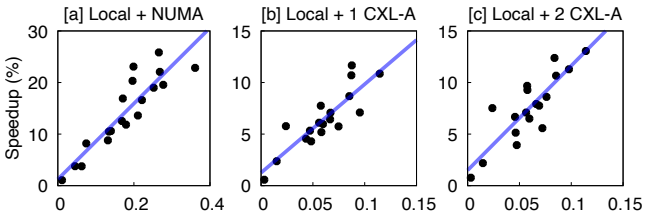

Figure 17: **Best-shot interleaving model for zNUMA, 1 and 2 CXL-A device(s).** *(a)-(c) show that best-shot interleaving model is accurate for both zNUMA and real CXL devices. Under 20 workloads in zNUMA, 1 and 2 CXL-A devices, best-shot interleaving can accurately predict and achieve performance gains of 2-21%, 1-13%, 1-26%, respectively.*

Finding #5:

1. Non-bandwidth-bound workloads typically cannot benefit from (weighted) interleaving. Even for bandwidth-bound workloads, we observed various slowdowns (and occasional speedups) across different interleaving ratio settings, reflecting the combined impact of CXL latency and bandwidth.
2. Various bandwidth-bound workloads have different optimal interleaving ratios, outperforming local DRAM performance to the greatest extent (there may exist a range of ratios yielding superior performance than local DRAM).

Our model aims to (1) predict whether a workload can benefit from interleaving (otherwise referring to a tiering policy), (2) predict the best interleaving ratio in one run, and (3) predict potential performance gains.

6.2 Best-Shot Interleaving Prediction

Our slowdown breakdown method (§4) can be used to analyze and predict NUMA interleaving performance as well. For those workloads benefiting from interleaving, the performance improvement (*i.e.*, negative slowdown) can still be attributed to various sources (DRAM, cache and store).

We observe that offcore latency goes down when workload performance is improved under efficient interleaving. Under effective interleaving, we observe offcore latency (L) and memory metric (M, §5.6) are complementary, where M indicates the latency-impact of the memory subsystem and offcore latency indicates memory bandwidth constraints. Thus, our model simply adopts ($L \times M$) as the performance metric (R) for interleaving performance prediction. Similarly, it is beneficial to break down the NUMA performance improvement or slowdown into various sources for accurate modeling. As such, we define the following models to predict interleaving speedup/slowdown from DRAM, cache and store, respectively.

$$R_{DRAM} = M_{DRAM} \times L_{DRAM} \quad (6)$$

$$R_{cache} = M_{cache} \times L_{cache} \quad (7)$$

$$R_{store} = M_{store} \times L_{store} \quad (8)$$

We will demonstrate how R can be used to predict the optimal NUMA interleaving performance across various NUMA interleaving ratios. Through an analysis of offcore latency when running all workloads on the local DRAM, we identify 20 bandwidth-intensive workloads (comprising 3 SPEC workloads and 17 Llama workloads) that exhibit performance improvements with NUMA interleaving. We use them for our model evaluation.

Evaluation on SKX-zNUMA. Figure 16a&b show our model accuracy to derive interleaving speedup contributions from DRAM and cache. We also observe that for workloads with over 5% improvement, the optimal ratio falls within the range of 34% to 40% (Figure 16c&d). Within this range, NUMA interleaving performance are roughly equally good. SKX2S local and zNUMA bandwidth ratio is approximately $5/3$ (Table 1). The theoretically ideal proportion of memory allocation is $5/8$ (62.5%) and $3/8$ (37.5%) for local and CXL, respectively. Consequently, the optimal ratio consistently falls within the range of 34% to 40% for workloads that significantly benefit from NUMA interleaving. For workloads with less pronounced benefits (<5%), the ratio predominantly ranges between 20% and 36%. The store model (R_{store}) is trivial for most workloads, thus, we omit store interleaving slowdown analysis here.

Evaluation on CXL-A. Figure 17 demonstrates the linear relationship between the best interleaving ratio and our predictor (R). Compared to Caption [48], our approach greatly simplify the process in an automatic way. Overall, best-shot interleaving achieves 1-13% performance improvement compared to local-DRAM on 20 workloads (upper bound is limited by the relatively low CXL bandwidth).

Latency-bound workloads. For workloads not constrained by bandwidth, the performance varies approximately linearly across different ratios. Although we could not gain interleaving benefits, interestingly, our model still allows us to predict the slowdown under a givn interleaving ratio x using a simple mode such as $x \times \sum_{DRAM,cache,store} S_i$, where x represents $N/(N+M)$ (with N denoting the remote node ratio and M representing the local node ratio), and S_i denotes the slowdown on CXL for different hardware components.

Implication #4: Weighted page interleaving can be used to improve performance for certain bandwidth-bound workloads under local and CXL memory. However, the optimal interleaving ratio varies across different workloads and the degree of performance improvements also differs. Best-shot interleaving can help predict the best interleaving ratios to achieve optimal performance and predict the precise amount of performance gains.

Recommendation #3: For bandwidth-bound workloads, the users can rely on our best-shot interleaving policy to run their workloads using the best setup for optimal performance.

6.3 Tiering Characterization

We now show SupMario root-cause breakdown analysis and performance models can be applied to tiering systems to dissect inefficiencies in tiering systems.

Existing tiering designs implicitly treat each LLC miss equally in terms of their contribution to system performance and heavily rely on LLC misses as the primary technique for sampling hot pages as migration candidates. However, our slowdown breakdown analysis (§4) has demonstrated that LLC misses (or their rate, *i.e.*, bandwidth) cannot reliably serve as a performance predictor/metric. This is because LLC misses caused by prefetching or RFO may not directly impact system performance. For instance, a prefetched cacheline may end up not being used. Instead, we assert that the rate of LLC stalled cycles and other stall cycle-related events are more accurate measures to gauge and predict system pressure.

Nonetheless, current tiering policies overlook this nuance and indiscriminately assume that high rates of LLC misses (or equivalently, DRAM traffic) inevitably result in performance degradation, inadvertently promoting excessive pages to local DRAM. This approach carries two downsides. First, migrating a large number of pages incurs non-negligible overheads, further compromising workload performance. Second, the assumption that these pages merit promotion to the fast tier (local DRAM) is unfounded, as they may not induce significant slowdowns. In combination, these factors lead to suboptimal tiering performance.

Characterizing tiering inefficiencies. We now use the prior analysis to reason about potential inefficiencies in tiering systems with a realistic workload, namely tc-twitter. In Figure 18a, we applied our "slowdown" prediction models to analyze tc-twitter slowdowns-over-time under CXL. Here, we apply our model to a period of the workoad executions (*e.g.*, every 1B instruction interval). Similar to previous workload level DRAM-contributed slowdown prediction, applying the LLC-stalls together with MLP factor delivers better predic-

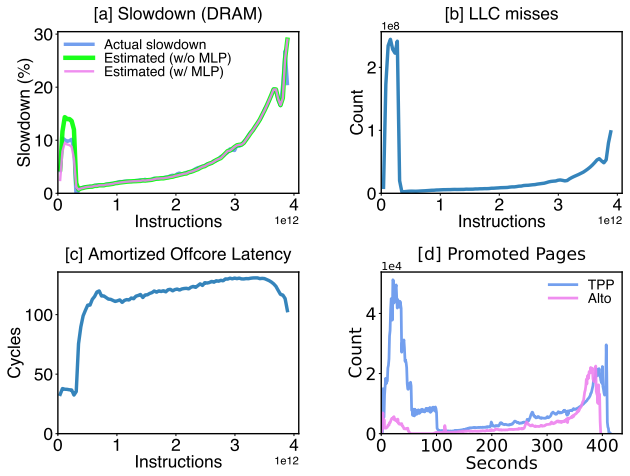

Figure 18: **Tiering performance characterization.** *LLC miss is not a good predictor for guiding memory tiering decisions. Our* `L3-stalls+MLP` *metric is more accurate. Note that over 99% of tc-twitter's slowdown originates from DRAM.*

tion, even at very fine-granularity (pink line is very close to blue line). We can see that the most significant DRAM-introduced slowdowns for tc-twitter occur during the final phase of the execution (3rd-4th billion instruction period). Upon further profiling of the LLC miss rate of tc-twitter over time (Figure 18b), we found that the bulk of LLC misses occur during the initial phase. That said, the substantial number of LLC misses during the initial phase do not contribute significantly to the workload performance as the rest of phases.

However, existing tiering designs operate under the assumption that performance degradation correlates positively with memory access rates. Consequently, they tend to aggressively "detect/scan" and migrate "hot" pages (both promotion and demotion).

Finding #6: As a result, it causes two potential problems: (1) "hot" pages are wrongly detected, *i.e.*, the migration of these seemingly hot pages does not lead to an overall performance enhancement as they don't cause CPU stalls by default; (2) As a result of the wrong hot page detection, it triggers unnecessarily high number of page migrations, which inversely degrade system performance (page-level migrations are long-latency and blocking operation in nature, causing high overhead). Combined, these would render tiering systems underperform compared to no-tiering.

Using TPP [42] as an illustrative example, we demonstrate how such memory tiering designs can exacerbate overhead and result in wrong page promotion decisions. In Figure 18d, the blue line shows the page promotion rate over time, which shows similar patterns as the LLC misses over time in Figure 18b. Correspondingly, a peak of 50,000 pages/s were observed around time 30s.

Finding #7: We define a new metric called "amortized offcore latency" considering both memory latency and MLP impact to capture the impact of CXL memory accesses to workload performance (details ommitted). And we find it to be able to capture workload performance very well.

In Figure 18c, we show that the "amortized offcore latency" during the initial phase remains notably low, indicating significant read request overlappings during the period. This overlapping mitigates performance degradation even in the presence of high LLC miss stalls, as many memory accesses, despite being affected by increased remote memory latency, are concealed by other parallel reads.

Further validation in Figure 18a and Figure 18b confirms that the high LLC misses during the initial phase result in marginal DRAM slowdown that is not as pronounced as observed during the final phase of the workload.

6.4 Alto: Adaptive Layered Tiering Orchestration

Our optimization is straightforward: limiting page promotions when the overlapping effect of memory accesses is evident. To this end, we propose **Alto, an adaptive layered tiering orchestration** scheme, built on top of TPP, to demonstrate the efficacy of our method. We chose TPP as it is the latest tiering effort tailored for CXL while alternatives like Hemem [45] and Memtis [39] primarily target persistent memory. Additionally, it's worth noting that page sampling (*e.g.*, Intel PEBS), an enabling technique for Hemem and Memtis does not support CXL yet.

We implement Alto by constraining the page promotion rate proportionally to the "amortized offcore latency" based on two thresholds. Specifically, if the "amortized offcore latency" (§5.3) falls below a lower bound, *e.g.*, 40 cycles, we disable page promotion to account for the evident memory access overlapping effect. Otherwise, if it exceeds the upper threshold, *e.g.*, 100 cycles, we do not limit page promotions. Both the lower bound and upper bound thresholds can be derived offline using a microbenchmark similar to §5.6.

In between, we gradually reduce page promotion rate as amortized offcore latency decreases, using a default 5-step interval. In our implementation, we achieve this by periodically ignoring potential promotion page candidates within small sets of pages. For instance, if we aim to allow 20% of TPP-identified candidate pages to be promoted, we allow the first two pages of every 10 pages to go through.

To monitor the "amortized offcore latency", we collect PMU counters periodically, *e.g.*, every 1s. Subsequently, we calculate the amortized offcore latency based on these counters, enabling us to dynamically adjust the page promotion rate based on the observed latency. Our user-level tool is lightweight and imposes no additional overheads. The kernel side only involves ~30 LOC changes to Linux MM migration policies. Reading a couple of PMU counters is extremely lightweight. Alto reads only 5 PMU counters every second, imposing almost zero overhead.

Alto Evaluation. We test Alto with 8 workloads, includ-

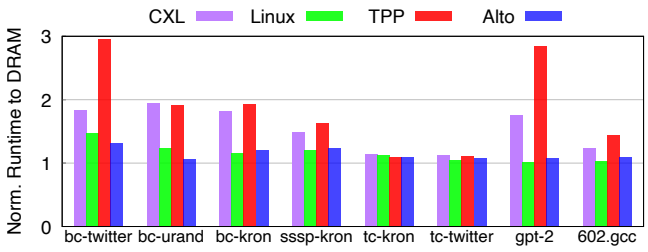

Figure 19: **Alto performance vs. TPP.** *X-axis is 8 different workloads under test, Y-axis is normalized workload runtime to Local DRAM. Alto can outperform TPP by 0.7-177%.*

ing graphs, ML and SPEC, comparing it with TPP and three additional settings: workloads backed by all local memory (Local), CXL memory (CXL), and default Linux hybrid local/CXL memory without tiering (default Linux). Since TPP performance is sensitive to the fast-tier memory size, we configure the local memory size to be large enough to accommodate the entire workload working set (profiled offline).

Workload working set (WSS) means the part of memory footprint which is actively accessed during workload runtime. We estimated WSS using heatmaps obtained via offline PEBS-based LLC-miss sampling (high sampling rate at 100 for accuracy). For each workload, we set its local DRAM to be slightly larger than its working set size (WSS), and CXL is used for the remaining memory footprint (*i.e.*, total memory footprint minus WSS). CXL memory is constantly accessed by the workloads as first-touch doesn't guarantee all the hot pages (in WSS) are initially placed in local memory, Figure 18d showcases heavy page promotions from CXL to local memory for tc-twitter. The gap between Linux and Local in Figure 18 stems from the accesses to CXL. We argue our local/CXL setup is fair to evaluate TPP as TPP performs much worse when more (slow) CXL memory is used, under which case Alto can actually improve TPP up to 9× (not shown).

In our evaluation, TPP typically underperforms default Linux due to erroneous page migration decisions and the resulting excessive overhead. We present all the results in Figure 19. Alto demonstrates improved performance compared to default Linux for workloads such as bc-twitter (+16%), bc-urand (+18%), and tc-kron (+3%). This enhancement stems from the fact that memory tiering can achieve better performance when it migrates correct pages. Alto enables TPP to constrain unnecessary page migrations by using an accurate performance metric, thereby aligning its behavior more closely with optimal performance scenario.

In detail, Alto demonstrates a performance improvement over TPP ranging from 0.7% to 177.5%. The most notable enhancement is observed in workload GPT-2, attributed to its highly parallel memory accesses and the substantial migration overheads in TPP. For bc-twitter, TPP even exhibits a 62% slower performance compared to CXL, while Alto significantly enhances TPP's performance. Workload tc-kron experiences the least performance improvement under Alto, primarily because only a small portion of it exhibits overlapped memory accesses. Alto outperforms Linux for 3 out of the 8 workloads in Figure 19 by 3%, 11%, and 14% while only slightly underperforming by 3-6% for the rest. Note that, in most cases, tiering designs such as TPP/AutoNUMA lose to first-touch/Linux as tiering becomes more sensitive to page migration overhead given the small latency gap (1.9-2.4×) between CXL/local memory.

It is an unfortunate (and maybe surprising) fact that first-touch/Linux under CXL is actually better than many (if not all) state-of-the-art tiering policies. According to our evaluations, TPP, AutoNUMA, and Nomad [52] loses to Linux by up to 181%, 22%, and 50%, respectively. Nomad authors also acknowledged in their paper (Section 4.2) that No-Migration (aka, Linux) performance exceeds (all) tiering solutions. This is because CXL latency is only 1.9-2.4× that of local-DRAM (for CXL-A,B,D) and the overhead of page migration can easily outweigh its benefits if migration policy is not carefully designed.

Implication #5: More broadly, we think tierability needs to be revisited in the CXL era. Alto's advantage over Linux/First-touch (even just) for some workloads calls for the need for principled approaches like ours to **(1)** diagnose and characterize tiering inefficiencies beyond hot/cold separation, and **(2)** revisit tiering policies designs to reduce migration overheads and focus on migrating performance-sensitive pages.

We utilize Alto to demonstrate how a performance metric from SupMario insights can significantly aid in identifying inefficiencies and enhancing existing tiering system performance with minimal changes. While Alto does not directly address the challenge of accurately sampling the most performance-critical hot pages for migrations, orchestrating the page migration rates indeed helps mitigate the overhead of incorrect migrations across a range of workloads. Additionally, we believe that SupMario's CPU-stall-based approach could further improve hot page sampling accuracy.

7 Discussion

SupMario implications. While the study primarily focuses on CXL devices, the high prediction accuracy on zNUMA indicates a pathway to performance observability, explainability, and predictability of general memory systems, relying solely on simple combinations of lightweight performance counters. Stemming from an offline performance breakdown analysis, SupMario performance models turn out to be workload-independent, accurate, robust, lightweight, simple, universal, and explainable. Our models are validated across 4 different CXL devices and 4 processor platforms, demonstrating the broad applicability of our model and the effectiveness of our modeling methodology. This paves the way for potential generalization. The simplicity of SupMario models should facilitate both offline and online usage. Our

performance models can potentially serve as general performance metrics/predictors for various tasks, such as workload/VM resource management and task scheduling. SupMario identifies key performance metrics that we envision can guide numerous system task optimizations, including hybrid memory policies integrating the benefits of interleaving and tiering, as well as new tiering policy designs such as improved hot/important page sampling.

CXL performance predictability. Our prediction models' deterioration from zNUMA to CXL-A or CXL-B indicates that CXL-B's worse tail latency also corresponds to the reduced predictability of our corresponding performance prediction models compared to zNUMA and CXL-A. This trend may worsen when future CXL-attached persistent memory or NAND Flash devices emerge. Addressing this challenge requires collaborative efforts from CPU, CXL device vendors, and OS/software developers to build QoS-aware and tail-tolerant software and hardware memory systems.

Additionally, CXL tail latencies also adversely affect academic CXL research based on emulation/simulation, such as zNUMA, given the current scarcity of CXL devices. Properly modeling and simulating CXL's intricate performance characteristics are essential to ensure a true reflection of real hardware characteristics.

Workload co-location: We validate that our models work for colocated applications as well (*e.g.*, multiple instances of various CPU 2017 workloads).

Future-Proofing. Future CXL devices will significantly improve bandwidth and somewhat improve latency. We anticipate our major indings and optimizations to remain valid.

1. CXL tail latencies are likely to persist due to various performance-functionality trade-offs in CXL controller implementations/optimizations, such as request scheduling, thermal management, QoS, and Reliability, Availability, and Serviceability (RAS). For instance, PCIe 6 will require thermal throttling, which could potentially worsen tail latencies [14, 15]. Additionally, with future CXL devices connected through CXL switches, the additional hops and potentially slower media (PM/Flash) will further increase the chances of latency unpredictability.
2. Future CXL workload slowdowns will likely be smaller than those in Figure 4a. Increased CXL bandwidth will benefit bandwidth-bound workloads, alleviating the 2-6× slowdowns seen in Figure 4a due to low-bandwidth per CXL device in our setup. Further latency reductions will improve the performance of latency-sensitive workloads, such as cloud applications, approaching NUMA performance. This is already evident with CXL-D* (hardware-interleaving across two CXL-Ds, >100GB/s bandwidth, green line) in Figure 4a, where bandwidth is no longer a bottleneck, similar to NUMA (black line). However, the latency gap between CXL and local memory persists. Mitigating slowdowns from CXL latencies will remain challenging without software/hardware optimizations, underscoring the need for detailed studies to characterize, analyze, model, and optimize performance to match local DRAM.
3. Our performance modeling approach will remain valid with improved CXL performance, and we expect our CXL prediction models (§5) to become more accurate, approaching the accuracy of zNUMA.
4. Our best-shot interleaving policy can further benefit bandwidth-intensive workloads such as HPC applications, by enabling them to exploit the higher aggregate system memory bandwidth.
5. We expect Alto to be more effective compared to state-of-the-art tiering policies, as their migration overhead will become more apparent when the latency gap between CXL and local memory narrows. For instance, our Alto evaluations on zNUMA (ideal-CXL) show an improvement to TPP up to 248% (not shown in the paper), significantly higher than the 177% improvement for Alto on current real CXL.

8 Related Work

CXL-based memory disaggregation. Memory disaggregation [28, 29, 39, 41, 42, 45, 51, 53] is a promising technique to improve memory resource utilization, which recently becomes more practical thanks to CXL's cache coherent interface. CXL-based systems [41, 57] need to address various aspects of memory management, including performance predictability. Our large-scale study contributes to a deep understanding of CXL performance implications, potentially motivating tailored management schemes to align with CXL performance characteristics for its imminent deployment.

Memory characterization. While DRAM characteristics have been extensively studied and modeled [31, 32, 34, 46, 55], the introduction of CXL prompts a reevaluation due to its unique performance characteristics. For instance, we unveiled CXL tail latency in the range of 100s of nanoseconds which is much larger than DRAM chip-level latency variations. Caption [48] is one of the first works characterizing real CXL devices, revealing measurement results of microbenchmarks and Redis/DLRM-like workloads. Due to the black box nature of CXL devices, Caption's analysis of workload performance is heavily reliant on speculations. While facing similar challenges, we purposely focused on different goals in our work: a much larger set of offline workload characterizations to reveal the detailed CXL impact on CPU pipelines, validated to be accurate, which further enabled us to develop an accurate performance prediction model. This model can be used for CXL memory management optimizations in interleaving and tiering scenarios. Our finding on CXL tail latencies, to the best of our knowledge, is a first in the community, and we carefully designed experiments to quantify its impact. Caption also con-

tributes an algorithm to derive a good interleaving ratio for bandwidth-bound workloads; however, Caption relies on a heuristic approach that requires running the workload multiple times (*e.g.*, 4–10 repeated runs) to converge on the result by relying on empirical metrics (*e.g.*, L1 miss latency). Our best-shot interleaving policy is inspired by Caption design and shares similar goals. However, we achieve more ambitious goals to predict both the optimal performance and weighted interleaving ratio in one run, guided by a systematic reasoning which is more accurate.

Memory tiering. Memory tiering [38, 39, 42, 45, 52, 56] typically relies on page table scanning, NUMA page-fault hints, and hardware event sampling (*e.g.*, Intel PEBS) to detect hot/cold pages, treating all memory accesses to DRAM equally without considering their relative contribution to workload performance in terms of CPU stalls. Although our work is not a typical tiering paper, our prediction models are shown to be useful in understanding inefficiencies in tiering and enhancing its performance. We hope that our findings and insights will guide the development of next-generation tiering policies, as we have demonstrated using the case of Alto in §6.4.

Performance prediction: Effective performance predictors, whether based on heuristics or machine learning, are crucial for system resource management and scheduling decisions. TMO [51] utilizes the PSI metric to guide tiering choices across multiple types of memory backends, measuring the amount of lost work due to resource shortages. Pond [41] employs an ML-based latency-sensitivity predictor to guide pool memory allocations. Caption [48] combines three metrics: L1 miss latency, DRAM latency, and IPC, to converge on the best NUMA interleaving ratio progressively. Our work shares similar aspirations but aims to identify a fundamental performance metric that is thoroughly reasoned and validated to be accurate. The novel combinations of a few performance counters in SupMario make it simple and lightweight. We believe our work is complementary to parallel explorations of new performance prediction methods with many potential use cases. For example, SupMario models could potentially serve as a simple and accurate replacement, *e.g.*, for Pond's [41] ML models, due to their simplicity and high accuracy.

9 Conclusion

In this paper, we present SupMario, the largest-scale CXL memory performance characterization conducted on a combination of hundreds of real-world applications and multiple hardware CXL and memory configurations. Our study unveils new findings regarding CXL performance characteristics, contributing novel insights to the community. Importantly, the characterization results enable a root-cause analysis for sub-µs memory latencies, leading to our most significant contribution: memory system performance prediction models built on just over ten performance counters. We demonstrate that our approach to derive the model and the model itself are useful in real-world interleaving and tiering scenarios. We plan to open-source SupMario and hope to inspire more research in this direction to better understand and manage CXL implications for efficient system designs.

10 Acknowledgments

We thank Yuyue Wang, Hansen Idden, and Shoaib A. Qazi for their assistance in setting up the experimental environment and conducting some of the initial experiments. This work was supported by funding from the NSF (grant numbers CNS-2339901 and CNS-2312785), as well as gift and contract funding from Samsung.

References

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

[2] Better Support for Locally-attached-memory Tiering. https://lwn.net/Articles/974126/.

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

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

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

[6] Exceptional Scalability with CXL Memory: Samsung and Red Hat Expand the Ecosystem. https://semiconductor.samsung.com/news-events/tech-blog/exceptional-scalability-with-cxl-memory-samsung-and-red-hat-expand-the-ecosystem/.

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

[8] Instruction Pipelining. https://en.wikipedia.org/wiki/Instruction_pipelining.

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

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

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

[12] Memory Hierarchy. https://en.wikipedia.org/wiki/Memory_hierarchy.

[13] NUMA Memory Policy. https://docs.kernel.org/admin-guide/mm/numa_memory_policy.html.

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

[15] PCIe 6.0's thermal throttling plans could slam brakes on performance. https://tinyurl.com/pciegen6-1.

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

[17] Questions from the Compute Express Link Exploring Coherent Memory and Innovative Use Cases Webinar. https://www.computeexpresslink.org/post/__q-a.

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

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

[20] Samsung CXL Solutions - CMM-H . https://semiconductor.samsung.com/us/news-events/tech-blog/samsung-cxl-solutions-cmm-h/.

[21] 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.

[22] 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/.

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

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

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

[26] Weighted Interleaving for Memory Tiering. https://lwn.net/Articles/948037/.

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

[28] Emmanuel Amaro, Stephanie Wang, Aurojit Panda, and Marcos K. Aguilera. Logical Memory Pools: Flexible and Local Disaggregated Memory. In *The 22nd ACM Workshop on Hot Topics in Networks (HotNets '23)*, 2023.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

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

[37] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[38] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] Vinicius Petrucci, Eishan Mirakhur, Nikesh Agarwal, Su Wei Lim, Vishal Tann, 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.

[44] 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.

[45] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[52] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[53] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[54] 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.

[55] 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.

[56] Huang Ying. AutoNUMA: Optimize Memory Placement for Memory Tiering System. https://lwn.net/Articles/835402/.

[57] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.