Performance Predictability in Heterogeneous Memory

Jinshu Liu
Virginia Tech
Blacksburg, USA

Hanchen Xu
Virginia Tech
Blacksburg, USA

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

Marcos K. Aguilera
NVIDIA
Santa Clara, USA

Huaicheng Li
Virginia Tech
Blacksburg, USA

Abstract

Heterogeneous memory combining DRAM and CXL exhibits variable performance, yet existing metrics correlate weakly with actual slowdown. We present Camp, a principled framework for predicting CXL-induced slowdown. Our key insight is that a DRAM run (plus a CXL run for bandwidth-bound workloads) exposes the causal microarchitectural pressure points where CXL latency translates into additional processor stall cycles. Camp captures these signals using 12 performance counters to analytically decompose slowdown into three orthogonal components: demand reads, cache/prefetching, and stores. Camp also introduces a closed-form model for software-based weighted interleaving that predicts performance across DRAM–CXL ratios. Across 265 workloads on NUMA and three CXL devices, Camp achieves 91–97% prediction accuracy within 10% absolute error. We demonstrate that these models enable practical system policies, including "Best-shot" interleaving and colocated workload placement, improving performance by up to 21% and 23% over existing tiering and colocation approaches.

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

Keywords: CXL Memory, Modeling, Prediction, Interleaving

ACM Reference Format:
Jinshu Liu, Hanchen Xu, Daniel S. Berger, Marcos K. Aguilera, and Huaicheng Li. 2026. Performance Predictability in Heterogeneous Memory. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790201

1 Introduction

Modern servers increasingly adopt heterogeneous memory architectures, combining fast local DRAM with larger, slower tiers such as NUMA and Compute Express Link (CXL) attached memory. While this disaggregation expands capacity cost-effectively, it introduces significant sensitivity in application performance. Misplacing latency-sensitive data onto slower tiers can trigger severe pipeline stalls, waste scarce DRAM bandwidth, and violate Service Level Objectives (SLOs) [18, 34, 36, 37, 46].

Effective management of these systems therefore requires *predictive visibility*: the ability to quantify exactly how a placement decision will impact application slowdown *before* that decision is enacted. Without accurate predictors, cloud operators are forced to overprovision DRAM to minimize risk, while runtimes and operating systems rely on reactive migration loops that waste resources correcting placement errors after performance has already degraded.

Despite extensive prior work, a gap remains between identifying performance signals and predicting slowdown. Profiling tools identify architectural events correlated with performance [26, 30, 36, 40, 56], but stop short of quantitative slowdown forecasts. Runtime systems employ heuristics, such as access frequency, LLC misses, latency, or stall cycles to guide tiered placement [24, 27, 38, 45, 51, 59]. However, these approaches are inherently *reactive* (detecting degradation post facto) or *descriptive* (attributing observed degradation to causes). For example, Melody [36] provides a robust framework for decomposing slowdown into components, but it is an *attribution* tool: it requires execution on both DRAM and CXL to explain the past. Similarly, SoarAlto [38] uses Memory-Level Parallelism (MLP) as a reactive metric to model demand-read-induced slowdown for tiering decisions, but does not offer a forward-looking model to predict overall slowdown or synthesize optimal interleaving ratios a priori.

This work introduces Camp[1], a framework that bridges this gap by transforming *offline attribution* into *prediction*. We address a fundamental question: *Can we predict how a workload will perform on CXL or under weighted interleaving across DRAM/CXL [16] using intrinsic workload signatures?* To answer this, we conduct a comprehensive analysis across 265 diverse workloads and four memory backends (NUMA and three CXL).

Our central insight is that slowdown on slower memory tiers (*e.g.*, CXL) is not an opaque property of the memory device, but a predictable consequence of microarchitectural

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790201

[1] Camp stands for "**C**ausal **A**nalytical **M**emory **P**rediction."

pressure points inside the CPU. We find that the structural bottlenecks driving CXL stalls, such as delayed data accesses in Line Fill Buffers (LFB), backpressure in Store Buffers (SB), and MLP-limited retirement, leave distinct "fingerprints" that are visible even during DRAM execution. These pressure points correspond directly to the three dominant sources of heterogeneous memory slowdown: demand reads, cache fills, and store-induced backpressure [36]. By isolating these signals, we construct analytical models that transfer DRAM-only measurements into accurate CXL slowdown forecasts. Because these models capture fundamental pipeline behavior, they generalize seamlessly across both pure CXL and interleaved DRAM–CXL configurations, enabling "what-if" analyses impossible with reactive metrics.

Building on these insights, Camp provides a lightweight, principled prediction framework using at most 12 Performance Monitoring Unit (PMU) counters. To predict CXL slowdown Camp models demand-read, cache, and store-induced slowdown components for non-bandwidth-bound workloads using a *single, DRAM-only execution*. Camp achieves 97% accuracy under NUMA and 91%–96% under CXL, with most predictions falling within a 10% error margin. For interleaving scenarios, Camp derives an analytical model that synthesizes performance curves for *any* DRAM–CXL ratio from at most two profiling runs.

We demonstrate the practical power of Camp through "Best-shot," a predictive interleaving policy and colocated workload placement. Unlike prior approaches that require iterative search or reactive migration, Best-shot analytically determines the optimal DRAM–CXL ratio. Best-shot improves performance by up to 21% over state-of-the-art tiering systems. Additionally, Camp-guided colocation scheduling improves performance by up to 23% over hotness-based placement by accurately modeling interference tolerance. In summary, this paper makes the following contributions:

- We introduce Camp, the first framework to predict all three sources of heterogeneous memory slowdown (demand-read, cache, store) from a single DRAM execution, enabling proactive "what-if" analysis prior to deployment.
- We derive explainable prediction models based on microarchitectural causes: prefetching inefficiency for cache slowdown, store-buffer backpressure for store slowdown, and latency/MLP dynamics for demand-read slowdown. The framework requires at most 12 standard PMU counters.
- We evaluate Camp on 265 workloads across three Intel microarchitectures and three CXL expanders. Camp achieves 0.97 Pearson correlation with actual slowdown, significantly outperforming prior metrics (0.37–0.88).
- We develop a predictive interleaving model that derives performance curves for *any* DRAM–CXL ratio using at most two profiling runs, obviating iterative search.
- We demonstrate Camp's versatility across two use cases. "Best-shot" analytically predicts the optimal interleaving ratio, outperforming existing tiering and interleaving policies. Camp-guided colocation scheduling improves performance by up to 23% over conventional placement.
- We open-source Camp artifacts, including benchmarks, datasets, and policies, at https://github.com/MoatLab/CAMP.

	MPKI	BW	Lat.	IPC	Stall	MLP	LFB/SB	Pearson
Memstrata [59]	√							0.40
X-Mem [23]					√			0.84
Colloid [51]			√					0.37
BATMAN [20]		√						0.66
Caption [46]			√	√				0.60
SoarAlto [38]			√		√	√		0.88
Camp (Ours)			√		√	√	√	**0.97**

Table 1. Performance metric comparison. *Pearson correlation with actual slowdown across 265 workloads on NUMA. Camp achieves 0.97 correlation by predicting all sources of slowdown using latency, MLP, hardware buffer pressure, and stall metrics. Prior approaches using subsets of these metrics achieve lower correlations (0.37–0.88).*

The rest of the paper is organized as follows. §2 reviews background and related work. §3 provides an overview of the Camp framework. §4 derives the core slowdown models. §5 develops the analytical interleaving model. §6 evaluates Camp in tiering and colocation use cases. §7 concludes.

2 Background and Related Work

Heterogeneous memory introduces a fundamental tension in modern system design [36, 37, 46, 47]. The resulting performance impact is highly workload-dependent: some applications tolerate this latency with minimal slowdown, while others experience severe pipeline stalls.

The central software challenge is therefore *predicting* which workloads can tolerate CXL and what fraction of their footprint can be placed there without incurring unacceptable slowdown. This section explains why existing metrics fail to provide such predictive capability, reviews the microarchitectural mechanisms that govern slowdown, and positions Camp relative to prior work. We use *slow memory/tier* and *CXL* interchangeably to refer to memory tiers with substantially higher access latency than local DRAM.

2.1 The Predictability Gap: Why Metrics Fail

Existing systems often rely on simple, metric-driven policies to guide tiering and interleaving decisions, aiming to identify latency-sensitive workloads or pages and to balance latency against aggregate bandwidth [33, 38, 42, 44, 45, 55]. In practice, however, commonly used metrics correlate poorly with actual CXL slowdown. Our analysis of 265 workloads confirms that these proxies can lead to incorrect placement and interleaving decisions (Figure 1, Table 1).

Access frequency (MPKI). Systems such as Memstrata [59] use misses per kilo-instructions (MPKI) to identify "hot" pages. While MPKI captures access intensity, it ignores *latency tolerance*. Workloads with high access frequency

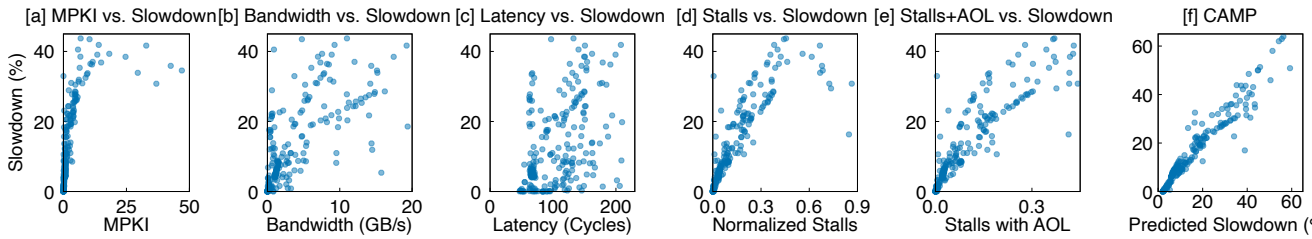

Figure 1. Correlation of common metrics and performance slowdown. *Figures (a)-(e) show that commonly used metrics, including MPKI, memory bandwidth, access latency, stall cycles, and AOL (defined as* $\frac{Latency}{MLP}$*) [38] exhibit weak correlation with CXL-induced slowdown across 265 workloads. In contrast, CAMP's predictor (f) shows a near-linear relationship with the observed performance slowdowns (more in §4).*

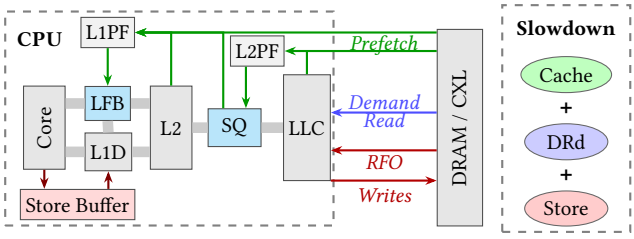

Figure 2. Slowdown breakdown. *Workload slowdown under CXL/NUMA decomposes into three orthogonal components: cache slowdown from late prefetch arrivals (S_{Cache}), DRAM slowdown from demand read stalls (S_{DRd}), and store slowdown from buffer backpressure (S_{Store}). Overall slowdown $S = S_{Cache}+S_{DRd}+S_{Store}$ [36].*

but abundant memory-level parallelism (MLP) can hide slow-memory latency and suffer less slowdown, whereas pointer-chasing workloads with moderate access frequency suffer disproportionately due to serialized memory accesses. Moreover, memory access frequency fails to account for store buffer backpressure and prefetch inefficiency.

Average latency and bandwidth. Approaches such as Colloid [51] and TierTune [32] monitor memory access latency. However, latency is a poor proxy for performance impact on modern out-of-order cores. Two workloads experiencing the same observed latency can incur vastly different stall cycles depending on instruction dependencies, reorder buffer occupancy (which limits the number of in-flight instructions), and overlap among outstanding misses, *etc*. Similarly, bandwidth provides imprecise signals. Moreover, neither latency nor bandwidth captures impacts on CPU data buffers.

Stall cycles. Approaches such as Intel Top-Down Analysis [56], PathFinder [35], X-Mem [23], and MSH [39] measure stall cycles directly. While stall cycles quantify current performance loss, they are insufficient for prediction. First, stall cycles measured on DRAM do not scale linearly with increased latency or reduced bandwidth on CXL; the scaling factor depends on impacts from multiple sources, such as overlap among outstanding requests, dependency structure, and offcore latency. Second, executing on a slower tier often changes workload behavior, altering prefetch timeliness, store buffer draining, and overlap patterns. Finally, aggregate stall metrics conflate demand reads, cache and prefetch effects, and store-induced stalls, even though these components respond differently to CXL. As a result, relying solely on stall cycles remains reactive and usually mispredicts slowdown. SoarAlto [38] partially addresses this limitation by amortizing latency with MLP in AOL=$\frac{\text{Latency}}{\text{MLP}}$ as a derived metric (Figure 1e), but it still fails to capture cache- and store-induced amplification effects.

Several systems augment these metrics using learning or control techniques [22, 34, 57, 58]. While effective in specific settings, these approaches still rely on empirical signals without a decomposed, causal model of slowdown, limiting their ability to generalize across memory configurations.

2.2 The Limits of Reactive Management

Without accurate predictors, heterogeneous memory relies on reactive management or expensive offline profiling.

Reactive tiering. Systems such as Nomad [55], Memtis [33], and TPP [42] migrate pages only after detecting hotness or performance degradation. This trial-and-error process incurs migration overhead and exposes workloads to performance loss before corrective action can be taken. ML-based systems such as Pond [34], Kleio [22], and ArtMem [58] improve robustness but still lack an explicit model that predicts the impact of placement decisions *a priori*.

Interleaving challenges. Weighted interleaving across DRAM and CXL can improve aggregate bandwidth [9, 29, 31, 37, 46, 53, 54], but determining the optimal ratio is non-trivial. Bandwidth-bound workloads can benefit from aggressive interleaving, while latency-bound workloads suffer severe slowdown. Without a predictive model, operators must rely on exhaustive search, leaving substantial performance potential unrealized.

2.3 Microarchitectural Pressure Points

Accurate prediction of CXL slowdown requires microarchitectural bottleneck reasoning as latency increases [21, 25]. Figure 2 illustrates the critical data paths in the CPU.

Fill buffers (LFB/SQ). Modern processors track outstanding cache misses using small hardware buffers rather than issuing requests directly to memory. These buffers hold outstanding misses from an upper-level cache until the corresponding cacheline arrives from the lower level [1, 5, 50,

52]. When multiple accesses miss on the same cacheline, they coalesce into a single entry, conserving space and avoiding redundant tracking. The Line Fill Buffer (**LFB**) tracks L1 misses, while the SuperQueue (**SQ**) tracks L2 misses to the uncore. These structures typically contain only tens of entries. As memory latency increases, it takes longer for prefetch requests to fill these buffer entries, causing additional stall cycles for cache-level data accesses. Moreover, requests from L1 and L2 prefetchers share LFB and SQ resources with demand accesses, and contention in these buffers can further incur cache-level stalls.

Store buffer (SB) backpressure. Stores are buffered asynchronously to allow execution to proceed, but Read-for-Ownership (**RFO**) requests to CXL significantly delay buffer draining. RFOs are mandatory read requests issued before each write to obtain exclusive cache-line ownership, causing stores to inherit CXL read latency. When the Store Buffer (**SB**) fills, subsequent stores block and stall the pipeline. This store-induced slowdown is largely invisible to read-centric metrics but becomes pronounced under CXL latency.

2.4 Camp vs. Melody and SoarAlto

Melody [36] showed that CXL slowdown can be decomposed into three additive components (Figure 2):

$$\text{Slowdown } (S) \approx S_{\text{DRd}} + S_{\text{Cache}} + S_{\text{Store}} \quad (1)$$

where S_{DRd} captures demand-read stalls, S_{Cache} captures cache/prefetch effects, and S_{Store} captures store backpressure.

While Melody establishes a powerful decomposition, as shown in Table 2, it is fundamentally an *attribution* technique: it requires executing workloads on DRAM and CXL to measure these components. Camp builds on this insight but pivots from attribution to *prediction*. The key observation is that DRAM-only execution already exposes sufficient microarchitectural signals such as buffer occupancy, prefetching inefficiency and latency–MLP relationships, to infer how each component will amplify under CXL.

By grounding prediction in these microarchitectural points, Camp enables "what-if" analysis without executing on CXL. Table 2 contrasts Camp with prior approaches. Unlike scalar metrics such as AOL in SoarAlto [38] or MPKI, Camp achieves substantially higher accuracy by modeling the causal mechanisms that govern slowdown, achieving 0.97 Pearson correlation versus 0.88 for AOL.

3 Camp Overview

This paper introduces Camp, a framework that fundamentally shifts heterogeneous memory management from *reactive observation* to *proactive prediction*. Camp relies on a single organizing principle: performance degradation on slow memory is not an opaque device property, but a predictable consequence of microarchitectural pressure points, *i.e.*, hardware bottlenecks where increased latency directly translates into pipeline stalls. By modeling latency's impact at the specific "pressure points" where it translates into additional execution cycles, Camp predicts workload performance on CXL using only DRAM execution signals.

	Camp (Ours)	**Melody** [36]	**SoarAlto** [38]
Goal	Prediction	Attribution	Tiering
Model	Analytical	Post-hoc	Reactive
Input	DRAM (+CXL)	DRAM + CXL	Live system
Coverage	S_{DRd}, S_{Cache}, S_{Store}	All 3 (post-hoc)	S_{DRd}-only
Pearson	**0.97**	N/A	0.88
Interleaving	✓	×	×

Table 2. Camp vs. related work. *Camp is a* prediction *framework that estimates all three slowdown components (S_{DRd}, S_{Cache}, S_{Store}) using 1–2 profiling runs, enabling proactive placement decisions. Melody is an* attribution *tool requiring both DRAM and CXL runs. SoarAlto uses a* reactive metric *to capture S_{DRd}-only effects.*

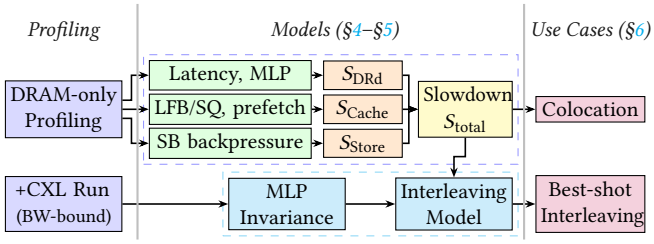

Figure 3. Camp framework overview. *DRAM-only profiling exposes PMU signals that predict per-component slowdowns (§4). For bandwidth-bound workloads, a additional CXL run enables interleaving prediction (§5). Both models feed practical use cases: colocation scheduling and Best-shot interleaving (§6).*

At a high level, Camp observes how a workload stresses the CPU pipeline on DRAM, predicts how those stresses amplify on slower memory, and composes the results to guide placement decisions. Figure 3 illustrates the Camp workflow. Unlike prior approaches that require iterative migration or exhaustive profiling, Camp operates as a feed-forward predictor. It extracts microarchitectural signatures from a standard DRAM-only run, transforms them into per-component slowdown forecasts, and synthesizes performance curves for arbitrary DRAM–CXL interleaving ratios. This enables operators to identify the optimal placement strategy *before* a workload is deployed. On 265 workloads across NUMA and three CXL devices, Camp achieves up to 0.97 Pearson correlation with actual slowdown, significantly outperforming metrics used by existing systems (Table 1). For cloud operators, this means CXL placement decisions can be made at job submission time, eliminating the runtime monitoring and migration overheads that burden reactive tiering systems.

3.1 Core Insight: From Attribution to Prediction

The state-of-the-art approach, Melody [36], is an *attribution* tool: it explains *why* slowdown occurred by comparing post-hoc measurements from both DRAM and CXL runs. While valuable for profiling, attribution cannot directly guide

placement decisions for new workloads.

Camp converts this decomposition into prediction. Our key insight is that the microarchitectural root causes of CXL slowdown, *i.e.*, pipeline stalls due to latency and buffer exhaustion, leave distinct fingerprints even when running on fast memory. We identify the following *DRAM-visible pressure points* that serve as precursors to CXL slowdown:

(a) Latency–MLP dynamics → S_{DRd} (§4.1). The interaction between MLP and access latency on DRAM predicts how demand-read stalls will amplify under higher CXL latency. Workloads with low latency-to-MLP ratios experience disproportionate slowdown because out-of-order execution cannot hide the extended memory access time.

(b) Fill buffer pressure → S_{Cache} (§4.2). High reliance on LFB (the buffer between L1 and L2 cache) and SQ (the buffer between L2 and LLC) during DRAM execution signals vulnerability to prefetch inefficiency on slower tiers. Late prefetches also cause demand reads to be delayed, when demand accesses are served from these transient buffers.

(c) Store buffer occupancy → S_{Store} (§4.3). Frequent cycles with a full Store Buffer indicate that RFO latency, the time to fetch a cacheline before writing, will become a serialization bottleneck on CXL. Higher RFO latency on CXL delays store buffer drain, blocking subsequent writes and eventually stalling instruction retirement.

(d) MLP invariance → interleaving performance curves. A critical enabler for *interleaving prediction* is our experimental observation that MLP remains approximately constant across interleaving ratios (§5.2.1). This stability occurs because MLP reflects the CPU core's ability to issue parallel loads, which depends on instruction-level parallelism rather than memory contention. With MLP approximately fixed, predicting performance across arbitrary DRAM–CXL ratios reduces to modeling how latency evolves as data is distributed between tiers (§5).

Because these pressure points reflect fundamental pipeline mechanics rather than device-specific latencies, predictors derived from a DRAM baseline through *(a)–(c)* can generalize to forecast behavior on CXL, NUMA, or interleaved configurations *(d)*.

3.2 The Camp Framework

Camp implements the insights through a three-stage pipeline for CXL and interleaving slowdown prediction in Figure 3:

3.2.1 Profiling. Camp collects a compact signature using only 12 PMU counters (in Table 5) during a single DRAM-only execution. These counters capture the intensity of the pressure points defined above. For bandwidth-bound workloads, a CXL profiling run captures the bandwidth saturation point needed to model interleaving contention.

- *Latency-bound workloads* require only a single DRAM-only run. Because these workloads do not saturate memory bandwidth, their behavior on CXL and interleaved DRAM–CXL setup can be fully inferred from DRAM measurements using the models in §4–§5.
- *Bandwidth-bound workloads* require two profiling runs for interleaving prediction (§5): one on DRAM and one on CXL. The CXL run is necessary because bandwidth saturation creates non-linear latency inflation, *i.e.*, contention interactions that cannot be inferred from DRAM alone.

3.2.2 Modeling. These signals feed two analytical engines.

- The *CXL slowdown model* (§4) maps each pressure point to its corresponding slowdown component. It predicts the absolute magnitude of S_{DRd}, S_{Cache}, and S_{Store} by modeling how stall cycles change under CXL latency.
- The *interleaving model* (§5) extends these predictions across arbitrary DRAM–CXL ratios under weighted interleaving policy. It synthesizes a continuous performance curve by modeling how latency evolves as data is distributed between tiers, exploiting MLP stability to reduce the problem to latency estimation (the non-linear interaction between bandwidth offloading and latency penalties).

3.2.3 Decision. The resulting predictions enable *analytical* optimization. Instead of searching for a good configuration, Camp calculates the "Best-shot" interleaving ratio and guides colocations that minimize system-wide stalls (§6).

3.2.4 Design challenges. Developing this framework requires overcoming three hurdles:

- *Inference without observation.* Unlike Melody, which measures CXL stalls directly, Camp *infers* future stalls without ever observing slow-memory execution. We achieve this by deriving transfer functions that model how increased latency affects specific microarchitectural structures.
- *Non-linear scaling.* Slowdown is not a linear function of latency. Camp captures component-specific non-linearities, such as the MLP and latency impacts on memory stalls, prefetcher inefficiency, and store buffer backpressure, by modeling the underlying microarchitectural causes.
- *Microarchitectural grounding.* To ensure interpretability, Camp avoids black-box ML as used in prior works [34]. All models are derived from microarchitectural reasoning grounded in established architectural principles, ensuring that every prediction can be traced back to specific hardware events (*e.g.*, LFB reliance).

3.3 Camp Use Cases

By decoupling prediction from execution, Camp enables capabilities previously unavailable to reactive systems:

Proactive interleaving ("Best-shot", §6.1). Existing tiering systems like Colloid [51] reactively migrate pages based on heuristics (*e.g.*, latency). Camp enables *Best-shot*, a policy that analytically calculates the optimal DRAM–CXL ratio. This allows systems to configure the optimal interleaving

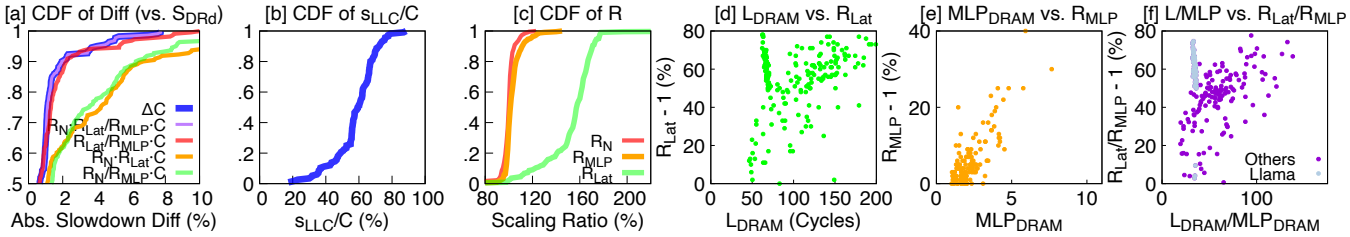

Figure 4. Inferring demand-read slowdown (S_{DRd}) from memory-centric proxies. *(a) reports the estimation error of different proxy metrics across workloads. Proxies based on memory-active cycles (C), combined with the latency and MLP scaling factors R_{Lat} and R_{MLP}, most closely track the observed demand-read slowdown. (b) shows the fraction of memory-active cycles stalled by last-level cache misses ($\frac{s_{LLC}}{C}$). (c) summarizes scaling ratios from DRAM to CXL. The number of concurrent requests R_N remains relatively stable, while latency and MLP scale more substantially. (d) and (e) show how baseline DRAM access latency and DRAM MLP relate to their respective scaling factors, R_{Lat} and R_{MLP}. (f) shows the relationship between baseline DRAM latency tolerance, measured as L/MLP, and the relative growth of latency over MLP on CXL.*

ratio immediately, avoiding the warm-up period and migration overheads of reactive tiering. Best-shot outperforms Colloid [51] by up to 21%, Soar [38] and Linux NUMA Balancing Tiering (NBT) [7, 8] by 17%, and Caption [46] by 5% on bandwidth-bound workloads (§6.2).

Interference-tolerant colocation (§6.3). Traditional hotness metrics (*e.g.*, MPKI) fail to predict CXL performance across workloads, while Camp provides reliable prediction even when interference exists. A workload with high MPKI may tolerate CXL well, while a low-MPKI workload may suffer disproportionately. Camp identifies latency-tolerant workloads that can be safely placed on CXL, freeing fast memory for latency-sensitive applications, enabling up to 12% better performance than MPKI-guided placement.

4 CXL Slowdown Prediction

In this section, we focus on modeling workload performance when running entirely on CXL, relative to DRAM. Camp translates the microarchitectural insights established in §3.1 into a quantitative prediction framework. Motivated by Melody [36], we adopt a divide-and-conquer strategy, modeling the three orthogonal sources of slowdown: demand reads (S_{DRd}), prefetching inefficiency (S_{Cache}), and store backpressure (S_{Store}), individually. The key innovation of Camp is deriving these models solely from DRAM-visible signals. By identifying PMU counters causally linked to stall behavior, we construct transfer functions that project DRAM measurements into CXL slowdown predictions.

Below, we detail the modeling methodology for each component (§4.1–§4.3) and then evaluate the framework's accuracy across diverse hardware platforms (§4.4).

4.1 Slowdown from Demand Reads (S_{DRd})

Demand-read slowdown (S_{DRd}) arises when demand read LLC misses cause additional pipeline stalls. Modeling this effect is non-trivial: not all cycles with outstanding requests appear as exposed stalls, and different microarchitectural factors interplay. Our goal is to (1) identify which factors drive stall growth, (2) quantify their impact, and (3) derive metrics that predict slowdown reliably.

4.1.1 Deriving the Predictor.

S_{DRd} originates from increased L3 miss stalls. We extend this term by focusing on **memory-active cycles (C)**, which counts cycles with at least one outstanding request, including both exposed stalls and cycles hidden by instruction retirement. Since instruction retirement cycles remain constant under increased latency, growth in C can reflect slowdown. Figure 4a (blue line) shows that ΔC-based slowdown closely matches measured S_{DRd}. Thus, S_{DRd} can be estimated as the increase in memory-active time normalized to total execution cycles (c):

$$S_{\text{DRd}} \approx \frac{\Delta C}{c} = \frac{C_{\text{CXL}} - C_{\text{DRAM}}}{c} \tag{2}$$

where C_{CXL} and C_{DRAM} represent C under CXL and DRAM. Based on Little's Law [43], C can be expressed as the function of the number of data requests (N), latency (L), and MLP:

$$C = \frac{N \times L}{\text{MLP}} \tag{3}$$

Substituting this into Eq. 2, the behavior on CXL is governed by how these three factors scale. We define the scaling ratios for latency, MLP, and N as: $R_{\text{Lat}} = \frac{L_{\text{CXL}}}{L_{\text{DRAM}}}$, $R_{\text{MLP}} = \frac{\text{MLP}_{\text{CXL}}}{\text{MLP}_{\text{DRAM}}}$, and $R_{\text{N}} = \frac{N_{\text{CXL}}}{N_{\text{DRAM}}}$. We then analyze their sensitivity and impact on slowdown, as illustrated by the purple, red, orange, and green curves in Figure 4a.

Step 1: Request stability ($R_{\text{N}} \approx 1$). First, we verify whether CXL latency alters the workload's request counts. Figure 4c plots the ratio of memory requests on CXL *vs.* DRAM (R_{N}). The distribution is tightly clustered around 1.0, confirming that for over 95% of workloads, N remains stable across DRAM and CXL executions. This indicates that N is primarily determined by program behavior and cache capacity, rather than memory latency. Thus, $N_{\text{CXL}} \approx N_{\text{DRAM}}$.

Step 2: Transfer function. With N constant, we substitute the scaling ratios into Eq. 2 to derive the transfer function:

$$S_{\text{DRd}} \approx \left(\frac{R_{\text{Lat}}}{R_{\text{MLP}}} - 1\right) \times \frac{C_{\text{DRAM}}}{c} \tag{4}$$

This equation exposes the physical mechanism: slowdown is driven purely by the *unhidden* portion of latency growth. If a workload scales its concurrency (R_{MLP}) to match the latency increase (R_{Lat}), the term becomes $(1-1)=0$, resulting in no demand-read slowdown. Figure 4a (red line) shows that Eq. 4 accurately estimates measured S_{DRd}, while ignoring R_{Lat} or R_{MLP} results in large deviations (green and orange lines). Eq. 4 shows that latency increase alone is insufficient to predict slowdown; only the portion of latency that remains unhidden after MLP scaling contributes to performance loss.

4.1.2 Modeling. Applying Eq. 4 is challenging because R_{Lat} and R_{MLP} cannot be directly inferred from a single DRAM execution. Predicting how a workload's (offcore) MLP and latency expand on CXL requires modeling the CPU's ability to tolerate additional memory latency. To address this challenge, we choose to explicitly model R_{Lat} and L_{DRAM}. Figure 4d shows workloads exhibit diverse baseline DRAM latencies because uncore or memory-controller buffers can better absorb requests, mitigating observed latency when hits occur. 78% of workloads have R_{Lat} between 140% and 180%, while the unloaded latency ratio for CXL versus DRAM is 156%. The 22% of workloads with lower baseline DRAM latency tend to exhibit smaller relative increases on CXL, indicating that greater buffering makes latency increases less pronounced. Figure 4d confirms this positive correlation between latency on DRAM and its increase ratio. Additionally, Figure 4c&e show the distribution of MLP increases on CXL, with 12% of workloads exhibiting increases exceeding 10%. The reason is that when concurrent requests are in flight, higher latency extends all pending requests, increasing the fraction of time spent at high concurrency. However, this scaling is bounded by static hardware resources such as miss-tracking queues (LFB/SQ).

The scaling factor vs. latency and MLP. In Figure 4f, we plot the observed scaling factor ($\frac{R_{\text{Lat}}}{R_{\text{MLP}}}-1$) against the workload's baseline latency tolerance ($\frac{L_{\text{DRAM}}}{\text{MLP}_{\text{DRAM}}}$). The data reveals a distinct pattern:

- **High L/MLP region:** High $\frac{R_{\text{Lat}}}{R_{\text{MLP}}}$ results from high L and low MLP. When MLP is low, workloads are serialized on DRAM. Thus, the scaling factor is dominated by the raw latency ratio (R_{Lat}).
- **Low L/MLP region:** Workloads with low L/MLP mostly tend to have high baseline concurrency. Therefore, the scaling factor is dominated by R_{MLP}, showing greater tolerance to latency increases.

The hyperbolic model. We model the latency tolerance factor $\frac{R_{\text{Lat}}}{R_{\text{MLP}}}$ as a function of baseline DRAM latency tolerance $\frac{L_{\text{DRAM}}}{\text{MLP}_{\text{DRAM}}}$. This asymptotic behavior is captured by a hyperbolic function of the form $f(x)=\frac{1}{p+q/x}$, which reflects diminishing returns as concurrency approaches hardware limits. The primary outliers are Llama workloads [6]. We fit this to the empirical data in Figure 4f to obtain platform-specific constants p and q, then substitute into Eq. 4 to derive the final predictor:

$$S_{\text{DRd}} \approx k \times \frac{1}{p+q\cdot\frac{\text{MLP}_{\text{DRAM}}}{L_{\text{DRAM}}}} \times \frac{s_{\text{LLC}}}{c} \tag{5}$$

where s_{LLC} (L3 miss stalls) serves as a proxy for C_{DRAM}. Here, s_{LLC} reflects baseline demand-read stall exposure. Figure 4b shows that the ratio of s_{LLC} to C ranges from 50–70% for 70% of workloads. Parameters p and q capture microarchitectural characteristics. k is a platform-specific scaling constant that converts the stall proxy s_{LLC} into memory-active cycles and is calibrated once per platform.

Outlier analysis. While the hyperbolic model fits the vast majority of workloads, Figure 4f highlights outliers, notably `Llama`. These workloads exhibit lower-than-predicted slowdown. This deviation likely stems from *burstiness*: our model uses average MLP, but AI workloads often alternate between computation and intense memory bursts. During bursts, instantaneous MLP may exceed the average, hiding more latency than the mean metric suggests.

4.1.3 Theoretical Alignment with Prior Art. It is noteworthy that the functional form of Eq. 5 resembles the AOL-based predictor used in SoarAlto [38]. However, a critical distinction exists in their derivation. SoarAlto identified the ratio L/MLP (AOL) as a predictive signal purely through empirical correlation analysis.

In contrast, Camp *derives* this relationship causally from Little's Law and the mechanics of memory-active cycle expansion. Our derivation explains the underlying reason AOL correlates with demand-read slowdown: it is not merely an empirical feature that correlates with performance, but the key factor for inferring how latency and MLP scale under CXL. By grounding the predictor in microarchitectural first principles, Camp provides an analytical explanation of the mechanism behind the metric.

Takeaway #1: Camp establishes that demand-read slowdown is predictable via a hyperbolic function of baseline DRAM latency tolerance (L/MLP), which generalizes across workloads and memory configurations. This formulation provides the theoretical ground truth for prior empirical metrics like AOL [38].

4.2 Slowdown from Prefetching (S_{Cache})

S_{DRd} captures stalls directly from demand reads. As memory latency increases, hardware prefetching mechanisms that supply caches and transient buffers are negatively impacted, causing additional cache stalls, termed S_{Cache}. Specifically, longer access latency reduces prefetcher timeliness. Extended pending time prevents prefetched data from filling into L1/L2 promptly, causing data that should be in L1/L2 to either be pending or, in the worst case, be dropped. Moreover, other types of data requests can be blocked when filling into transient buffers if contention exists. Melody [36] observed

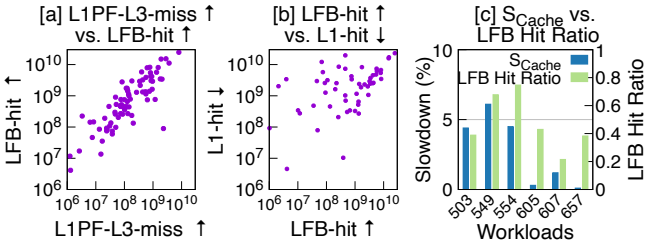

Figure 5. LFB pressure explains cache-induced slowdown. *(a) Increased L1 prefetches that miss L3 correlate with higher LFB hit counts; (b) Higher LFB occupancy is associated with reduced L1 hit rates, indicating backpressure on the cache hierarchy; (c) Workloads with larger cache slowdown tend to exhibit higher LFB hit ratios.*

that CXL latency degrades L2 prefetcher timeliness, causing up to 15% of workloads to suffer significant S_{Cache}. Recent studies further confirm that prefetching effectiveness and fill buffer behavior change substantially under high-latency memory [41, 49]. However, this observation leaves a causal gap: it does not explain *how* prefetch inefficiency translates into pipeline stalls. To predict slowdown, we must answer: (1) which specific structure becomes the bottleneck when prefetching falters, and (2) how do we quantify this pressure?

4.2.1 Reasoning.

We identify LFB as a critical pressure point. On CXL, the prefetching pipeline undergoes a distinct regime shift driven by the interaction between cache levels.

First, the L2 prefetcher fails to look far enough ahead to cover CXL latency, causing data to arrive too late for the L2. Consequently, the L1 prefetcher compensates by bypassing the L2 and issuing requests directly to memory via the uncore. CXL execution increases LFB hits, strongly correlated with increase of L1 prefetch L3 misses (Figure 5a), indicating more LFB-based data delivery. Correspondingly, L1D hits decrease with LFB hit increases (Figure 5b). Together, these observations suggest that data normally accessed from L1D is now accessed from the LFB due to delayed cacheline fills, caused by longer fetch time from lower memory levels under higher CXL latency. Since the L2 is inclusive of the L1, demand reads serviced by the L2 remain unaffected by CXL latency. Therefore, the delayed accesses observed in the LFB are driven primarily by these L1 prefetch loads fetching from memory. Longer memory latency extends pending time for in-flight L1 prefetch L3 misses. Additionally, increased L1 prefetch L3 misses (while L3 hits remain constant) indicate that expected L2 data now resides in memory. These combined effects cause L1 prefetcher-allocated LFB entries to experience longer fill time under increased memory latency.

Moreover, the extended occupancy on LFB can prevent other data accesses, including demand reads and writes, from allocating entries. This resource contention can also translate prefetch inefficiency into pipeline stalls.

4.2.2 Deriving the Predictor.

To model S_{Cache} based on causal factors, we quantify two aspects: the workload's reliance on the LFB for data delivery, and the fraction of that delivery sourced from prefetch requests to DRAM.

	SKX2S	SPR2S	EMR2S
CPU	Xeon 4110	Xeon 6430	Xeon 6530
Cores	10 @ 2.2GHz	32 @ 2.1GHz	32 @ 2.1GHz
LLC	14MB	60MB	160MB
DRAM	DDR4 2666MHz	DDR5 4800MHz	DDR5 4800MHz
DRAM Bandwidth	52/32GB/s	191/97GB/s	246/120GB/s
DRAM Latency	90/140ns	114/191ns	111/192ns

Table 3. Testbed. *We used three two-socket ("2S") servers with different CPU and memory configurations.*

Signal #1: LFB hit ratio ($R_{\text{LFB-hit}}$). The LFB-hit ratio captures workload dependence on the LFB for data access:

$$R_{\text{LFB-hit}} = \frac{\text{LFB-hits}}{\text{LFB-hits} + \text{L1D-misses}}$$

This ratio reflects two aspects: (1) the proportion of accesses that retrieve data from the same cacheline (*e.g.*, streaming patterns), and (2) the extent to which these accesses are served by the LFB rather than L1D. High LFB-hit ratios indicate strong LFB reliance, making such workloads more vulnerable to prefetch delays.

Figure 5c shows that high cache slowdown correlates with high LFB-hit ratios, but a high ratio alone does not guarantee severe slowdown, as LFB hits can result from prefetching, demand loads, and writes.

Signal #2: Prefetch-from-memory reliance (R_{Mem}). Therefore, a second metric quantifies the fraction of LFB allocations attributable to memory prefetches:

$$R_{\text{Mem}} = \frac{\text{L1PF-from-Memory}}{\text{Total LFB Allocations}}$$

A high R_{Mem} indicates that LFB allocations are primarily driven by prefetches from memory, making the workload sensitive to memory latency.

The predictor. We model S_{Cache} as the product of normalized cache stall cycles (s_{Cache}), the workload's reliance on LFB hits ($R_{\text{LFB-hit}}$), and its reliance on memory prefetches (R_{Mem}).

$$S_{\text{Cache}} \approx k \times R_{\text{LFB-hit}} \times R_{\text{Mem}} \times \frac{s_{\text{Cache}}}{c} \tag{6}$$

where k is a platform-specific calibration constant.

Takeaway #2: CAMP predicts S_{Cache} by modeling the impact of prefetching on cache stalls. The LFB-hit ratio ($R_{\text{LFB-hit}}$) and memory prefetch reliance (R_{Mem}) are the key factors to model increased cache stalls under CXL.

Having modeled the CXL impact on read paths (demand and prefetch), we next address the impact on the write path.

4.3 Slowdown from Stores (S_{Store})

Finally, we address the impact of CXL on the write path. Unlike loads, stores are typically asynchronous and removed from the critical path by the Store Buffer (SB). However, this buffering capacity is finite. We identify **SB Backpressure**

	CXL-A	CXL-B	CXL-C
DRAM	DDR4 2666MHz	DDR5 4800MHz	DDR5 4800MHz
Bandwidth	24GB/s	22GB/s	52GB/s
Latency	214ns	271ns	239ns
PCIe 5 Lanes	×8	×8	×16

Table 4. Three ASIC CXL 2.0 memory expanders. *Our CXL devices deliver 22–52 GB/s of bandwidth and exhibit latencies between 214ns and 271ns. CXL-C's nearly double bandwidth arises from its use of multiple memory channels.*

as the mechanism that converts hidden write latency into exposed pipeline stalls.

4.3.1 The Mechanism: RFO Serialization.

Every store operation requires the core to obtain exclusive coherence permissions for the target cacheline via a RFO request. While the core can issue the store instruction to the SB immediately, the SB entry cannot be freed until the RFO completes.

On CXL, RFO latency increases by around 2–3×. This creates a flow-control mismatch: the core issues store requests into the buffer (*i.e.*, SB) faster than the buffer can retire these requests. Once the SB fills, it asserts backpressure on the pipeline, potentially blocking the retirement of all subsequent instructions, even independent reads. Thus, under high write intensity, the "asynchronous" store mechanism degrades into a synchronous stall governed by RFO latency on CXL due to limited SB capacity.

4.3.2 Deriving the Predictor.

When the Store Buffer is full, the CPU pipeline stalls. This effect is amplified under CXL as increased memory latency exposes RFO delays to the CPU, creating pipeline backpressure.

To predict this, we measure the *Store Buffer Fullness Stalls* (s_{SB}) on DRAM. This counter captures the number of stall cycles during which the pipeline is back-pressured by a full SB. Since CXL extends the duration of each RFO, it proportionally extends the time the SB remains full. We therefore model store slowdown (S_{Store}) as a linear function of these stalls:

$$S_{\text{Store}} \approx k \times \frac{s_{\text{SB}}}{c} \tag{7}$$

where s_{SB} is the standard PMU counter for SB-full stall cycles and k is a platform-specific constant derived from microbenchmarks. This model captures the intuition that write-heavy workloads (*e.g.*, database logging or large memsets) are bottlenecked mostly by the rate at which the memory subsystem grants RFOs. By detecting SB saturation on DRAM, Camp identifies these workloads before migration.

4.4 Implementation and Evaluation

Camp translates the theoretical transfer functions derived above into a practical runtime predictor. This section details the deployment workflow, the mapping of abstract model terms to physical PMU counters, and a comprehensive evaluation across diverse hardware and workloads.

4.4.1 Workflow.

Camp operates in two phases:

#	Name	Brief Description
P_1[†]	STALLS_L1D_MISS	#*s* on L1 miss demand load
P_2[†‡]	STALLS_L2_MISS	#*s* on L2 miss demand load
P_3[†‡]	STALLS_L3_MISS	#*s* on L3 miss demand load
P_4[†‡]	L1_MISS	Load instructions missing L1
P_5[†‡]	LFB_HIT	Load instructions missing L1, hitting LFB
P_6[†‡]	BOUND_ON_STORES	#*s* where the Store Buffer was full
P_7[†]	PF_L1D.ANY_RESPONSE	All L1 prefetch requests to offcore
P_8[†]	PF_L1D.L3_HIT	L1 prefetch to offcore that miss L3
P_9	PF_L2.ANY_RESPONSE	L2 prefetch data reads, any response type
P_{10}	PF_L2.L3_HIT	L2 prefetch reads that hit in the L3
P_{11}	ORO.DEMAND_RD	Outstanding demand data read per cycle
P_{12}[†‡]	OR.DEMAND_RD	Demand data read requests sent to offcore
P_{13}[†‡]	ORO.CYC_W_DEMAND_RD	#*c* when demand read request is pending
P_{14}[‡]	LLC_LOOKUP.PF_RD	Cache & snoop filter lookups; prefetches
P_{15}[‡]	LLC_LOOKUP.ALL	Cache & snoop filter lookups; any request
P_{16}[‡]	TOR_INS.IA_PREF	Prefetch that misses in the snoop filter
P_{17}[‡]	TOR_INS.IA_HIT_PREF	Prefetch that hits in the snoop filter

Table 5. Intel PMU counters for Camp. *#c: number of cycles; #s is number of stall cycles;* ORO *(P_{11} and P_{13}) refers to* OFFCORE_REQUESTS_OUTSTANDING*;* OR *(P_{12}) refers to* OFFCORE_REQUESTS*;* LLC_LOOKUP *(P_{14} and P_{15}) refers to* UNC_CHA_LLC_LOOKUP*;* TOR_INS *(P_{16} and P_{17}) refers to* UNC_CHA_TOR_INSERTS*; Counters marked with* † *are used by SKX;* ‡ *by SPR/EMR in the models in § 4.4.3. Counters without symbols (P_9–P_{11}) are used during model derivation but cancel out and therefore do not appear in the final model. Including the cycle-count counter (omitted from the table), the SKX and SPR/EMR models use 11 and 12 counters, respectively.*

(1) **One-time calibration.** We run a lightweight suite of microbenchmarks to characterize the target hardware's behavior. This yields the platform-specific constants: the parameters (p, q) that define the CPU's MLP and latency impacts (Eq. 5), and the scaling coefficients (k) for each component model.

(2) **Runtime prediction.** During DRAM execution, Camp samples standard PMU counters (11 on SKX, 12 on SPR/EMR). These raw counts are fed into the calibrated models to forecast CXL-induced slowdown *a priori*.

Microbenchmarks. The calibration suite isolates specific pressure points: (1) *Pointer Chasing:* Isolates pure latency sensitivity (S_{DRd} with MLP≈1). (2) *Sequential Reads:* Drives high bandwidth to characterize MLP behavior. (3) *Strided Access:* Triggers prefetchers to calibrate S_{Cache} constants. (4) *Memset:* Generates back-to-back stores to characterize SB backpressure (S_{Store}).

4.4.2 Experimental Setup.

We validate Camp on three Intel microarchitectures: Skylake (SKX), Sapphire Rapids (SPR), and Emerald Rapids (EMR). To test generalizability, we use three distinct ASIC CXL 2.0 expanders (Table 3), denoted as **CXL-A**, **CXL-B**, and **CXL-C**. These devices span a range of latencies (214–271ns) and bandwidths (22–52GB/s). We also emulate a NUMA tier on SKX to validate the model's stability on standard multi-socket systems.

Workloads. We evaluate Camp on 265 workloads including SPEC CPU 2017 [13], PARSEC [19], GAPBS [17], PBBS [14],

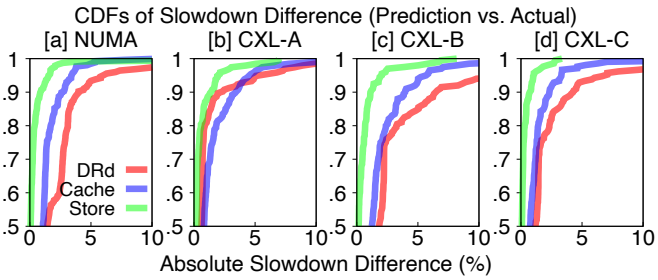

Figure 6. Prediction accuracy of individual slowdown components. *CDFs of absolute prediction error demonstrate that demand-read, cache, and store slowdown components are predicted accurately and consistently across SKX NUMA and three CXL devices.*

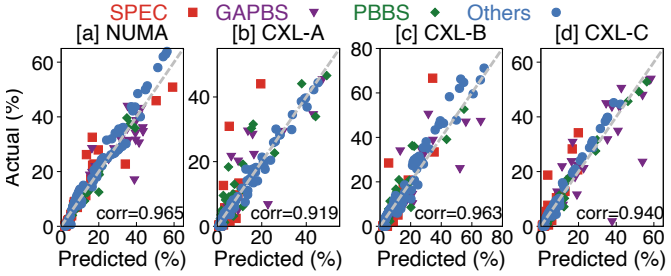

Figure 7. Overall slowdown prediction. *Predicted vs. actual slowdown for NUMA, CXL-A, CXL-B, and CXL-C across all workloads in (a)-(d), respectively.*

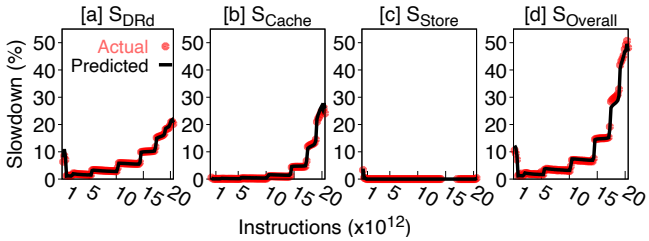

Figure 8. Time series prediction accuracy (`tc-kron`). *Actual vs. predicted slowdown overlap over time across slowdown all components.*

	NUMA	CXL-A	CXL-B	CXL-C
Pearson Coefficient	0.965	0.919	0.963	0.940
Absolute Error ≤ 5%	88.4%	88.7%	77.8%	92.4%
Absolute Error ≤ 10%	97.3%	94.3%	90.7%	96.2%

Table 6. Overall prediction accuracy. *CAMP predicted slowdowns closely match measured performance, with Pearson correlation coefficients between 0.919 and 0.965. Across configurations, 77.8%–92.4% of workloads have absolute error within 5%, and 90.7%–97.3% fall within 10%, indicating robust accuracy across diverse CXL devices.*

XSBench [48], Phoronix [10], and modern cloud applications (Redis [11], Spark [28], VoltDB [15], MLPerf [12], Llama [6], GPT-2 [3], DLRM [2]). This set covers a wide spectrum of memory behaviors, from pointer-heavy data structures to streaming analytics and AI.

4.4.3 Counter Mapping.

To deploy the models, we map the abstract terms (R_{Lat}, R_{MLP}, etc.) to specific Intel PMU counters (Table 5).

For S_{DRd}, we measure L and MLP using standard offcore request and cycle counters. For S_{Store}, we directly measure Store Buffer full cycles (P_6). For S_{Cache}, measuring R_{Mem} (the fraction of LFB fills from memory prefetches) is challenging because current PMUs lack precise data-source tracking for LFB allocations. We therefore approximate R_{Mem} using the ratio of offcore prefetch misses to total prefetch requests: $\frac{P_7-P_8}{P_7}$ on SKX and $\frac{P_{14}}{P_{15}} \times \frac{P_{16}}{P_{16}+P_{17}}$ on SPR/EMR. The overhead of reading these counters via Linux `perf` is negligible.

Final slowdown prediction model. $S = S_{\text{DRd}} + S_{\text{Cache}} + S_{\text{Store}}$, with PMU counters P_i, cycles c, and platform-specific constants k, p, q which are derived via microbenchmarks:

$$
\begin{aligned}
S_{\text{DRd}} &= k_{\text{drd}} \cdot \frac{P_3}{c} \cdot \frac{1}{p \cdot \frac{P_{12}}{P_{13}} + q} \\
S_{\text{Cache}}^{\text{SPR/EMR}} &= k_{\text{cache}} \cdot \frac{P_2 - P_3}{c} \cdot \frac{P_5}{P_4+P_5} \cdot \frac{P_{14}}{P_{15}} \cdot \frac{P_{16}}{P_{16}+P_{17}} \\
S_{\text{Cache}}^{\text{SKX}} &= k_{\text{cache}} \cdot \frac{P_1 - P_2}{c} \cdot \frac{P_5}{P_4+P_5} \cdot \frac{P_7 - P_8}{P_7} \\
S_{\text{Store}} &= k_{\text{store}} \cdot \frac{P_6}{c}
\end{aligned}
$$

4.4.4 Validation Results.

We evaluate CAMP by comparing its predictions against the measured CXL slowdown.

Accuracy by component. Figure 6 breaks down accuracy for each sub-model, showing the CDFs of prediction errors. CAMP achieves high precision across all components:

- S_{DRd} predicts within 5% error for **92–94%** of workloads on NUMA, CXL-A, and CXL-C. On CXL-B, accuracy is slightly lower (78.7% within 5% error) due to higher tail latency variance of the device, as reported by Melody [36].
- S_{Cache} predicts within 5% error for **93–97%** of workloads across all devices, confirming that our model ($R_{\text{LFB-hit}} \times R_{\text{Mem}}$) accurately captures prefetch inefficiency.
- S_{Store} predicts within 5% error for **93–97%** of workloads, validating the linear relationship between store buffer saturation and slowdown.

Overall accuracy. Figure 7 shows the correlation between predicted and actual total slowdown. CAMP achieves a Pearson correlation of **0.97** on NUMA and **0.91–0.96** on CXL. Notably, the model generalizes across devices: CXL-C provides 2× the bandwidth of CXL-A, and CXL-B has 27% higher latency than CXL-A, yet CAMP correctly predicts slowdown without requiring manual recalibration.

Misprediction analysis. While CAMP is highly accurate, ≈10% of workloads exhibit errors >5%. Analyzing these outliers reveals the boundaries of the model:

1. **Tail latency noise (underestimation).** Workloads with irregular access patterns (*e.g.*, graph analytics `pr-twitter`) often trigger worst-case tail latencies on CXL devices [36]. Since CAMP relies on average latency signals from DRAM, it underestimates the penalty when the CXL device's tail latency diverges significantly from its mean. This effect is

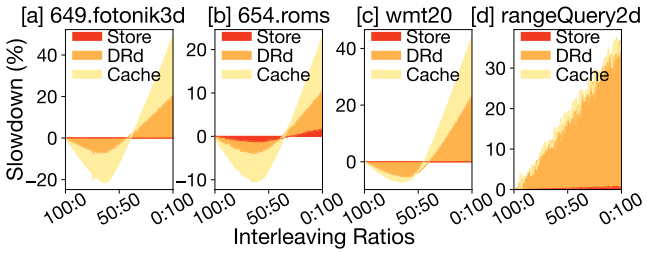

Figure 9. Weighted interleaving performance vs. ratios. *Per-component slowdown (S_{DRd}, S_{Cache}, and S_{Store}) under interleaving for* `649.fotonik3d`, `654.roms`, `wmt20`, *and* `rangeQuery2d`.

most pronounced on CXL-A and CXL-B.

2. **Hyper-parallelism (overestimation).** For workloads with extreme MLP (*e.g.*, `pr-kron` in GAPBS), the model tends to overestimate slowdown. This suggests that at very high concurrency levels, the CPU's ability to overlap latency scales non-linearly in ways that simple average MLP metrics do not fully capture.
3. **Instrumentation blindspots.** On SPR/EMR, the lack of precise counters for "L1 prefetch that hits in L2" forces us to use offcore proxies for R_{Mem}. This approximation can degrade accuracy for workloads with complex mid-level cache behavior (S_{Cache} errors).

4.4.5 Dynamic Prediction.

Real-world workloads exhibit phase behavior. Figure 8 demonstrates Camp's ability to track these dynamics. The predicted time series closely matches the measured slowdown per second, confirming that the causal modeling holds instantaneously, not just in aggregate.

Takeaway #3: Camp achieves >90% prediction accuracy across NUMA and CXL, validating that CXL slowdown is driven by fundamental microarchitectural bottlenecks (LFB/SQ/prefetch/SB) rather than opaque device properties.

4.4.6 Limitations and Future Work.

Camp's CXL slowdown model currently applies to regimes where device bandwidth is not saturated. Once bandwidth saturates, access latency can increase non-linearly, cascading into amplified demand-read (S_{DRd}), cache-induced (S_{Cache}), and store-induced (S_{Store}) slowdowns. Modeling these bandwidth-saturation effects from non-bandwidth-bound DRAM measurements is an interesting direction for future work.

Platform extensibility. Since Camp relies on counters that capture memory hierarchy efficiency relative to latency and bandwidth profiles, and does not depend on Intel-specific hardware features, we believe the approach can be extended to AMD and ARM platforms with equivalent PMU counters.

5 Synthesizing Interleaving Performance

§4 established a method to predict performance at the endpoints: 100% local DRAM and 100% CXL. However, modern systems increasingly employ *weighted interleaving* [9, 16], distributing pages across tiers to maximize aggregate bandwidth. This creates a continuous spectrum of performance possibilities between the two extremes. In this section, we extend our previous model to interleaved memory configurations and predict workload performance under a given interleaving ratio.

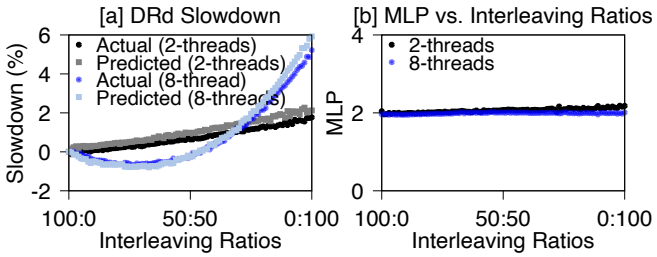

Figure 10. MLP and DRd slowdown (S_{DRd}) under interleaving (`603.bwaves`). *MLP remains mostly unchanged across different interleaving ratios. In addition to stall cycles, S_{DRd} can also be estimated using memory-active cycle count (P_{13} in Table 5).*

Challenges. Predicting performance at an arbitrary interleaving ratio (x) is non-trivial because shifting data creates a feedback loop: changing the ratio alters the traffic load on each tier, which non-linearly impacts contention and latency. Prior work [46] explores a limited set of ratios through multiple trial runs, often converging to suboptimal results.

Our approach. We propose a physics-based synthesis model. Our key insight is that while memory *latency* varies under interleaving, regardless of contention, the workload's MLP is structurally bounded and varies weakly with interleaving. By treating MLP as approximately constant, we derive a closed-form equation that predicts performance (*i.e.*, stall cycles) at any interleaving ratio as a function of how latency varies across interleaving ratios for both tiers, using only endpoint stall and latency measurements. This allows us to predict the full performance spectrum and the optimal interleaving ratio using at most two profiling runs.

5.1 Characterization: The Shape of Slowdown

We profiled 100+ workloads across 101 different interleaving ratios (100:0 to 0:100) to characterize the solution space and motivate our model. As shown in Figure 9, slowdown behavior falls into two distinct physical regimes:

- **Latency-bound (linear response):** Workloads not bounded by bandwidth (*e.g.*, `rangeQuery2d`) see no benefit from interleaving. Their slowdown increases linearly as more data is moved to the slow tier, since each remote access incurs an approximately fixed latency penalty.
- **Bandwidth-bound (convex response):** Bandwidth-intensive workloads (*e.g.*, `649.fotonik3d`, `654.roms`) exhibit a convex "bathtub" curve. There exists a specific optimal ratio where the gain from aggregate bandwidth (DRAM+CXL) outweighs the CXL latency penalty, often outperforming a pure DRAM-only configuration.

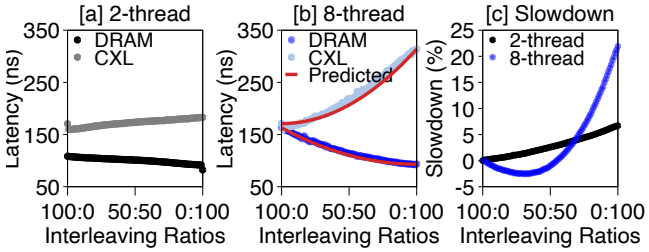

Figure 11. Latency and slowdown curves under DRAM–CXL interleaving (`603.bwaves`). *With 8 threads,* `603.bwaves` *is bandwidth-bound on DRAM, and interleaving improves performance, yielding negative slowdown near a 37:63 ratio in (c). Both local and CXL latencies follow parabolic trends. In contrast, the 2-thread configuration is not bandwidth-bound and exhibits nearly constant latency across interleaving ratios.*

These two regimes explain why interleaving is either harmful, neutral, or beneficial. Moreover, slowdown under interleaving can be attributed to sub-components (demand reads, cache, and stores), which makes it feasible to develop sub-models for each source. These observations motivate a general model that explicitly captures stall cycles variation across interleaving ratios, regardless of memory contention.

5.2 The Synthesis Model

Figure 10a shows that S_{DRd} closely matches ΔC-based slowdown. Therefore, our goal is to predict memory-active cycles $C(x)$ when a fraction x of the memory footprint is mapped to DRAM and $(1-x)$ to CXL under the weighted interleaving policy. We define the notation in Table 7. We use x as the interleaving policy knob (footprint fraction) and treat it as a monotonic proxy for the steady-state request fraction to the DRAM tier. Our experiments confirm that the fraction of data requests to the DRAM tier ($N(x)$) across interleaving ratios aligns well with the memory footprint fraction (x). For instance, 99% of data points show less than 2% absolute difference between tier request share ($N(x)/N$) and x for 8-thread `603.bwaves`. Thus, tier request share can be approximated by footprint share under weighted interleaving.

5.2.1 The Invariant: MLP Consistency.

Eq. 3 gives the relationship among N, L, and MLP. As N and L change with interleaving ratios, how does MLP behave? Our key observation is that *MLP is structurally bounded and varies minimally across interleaving ratios.*

- **Observation:** As shown in Figure 10b, whether a workload is bandwidth-bound (8 threads) or not (2 threads), the measured MLP per core fluctuates negligibly ($\leq$ 5%) as we sweep the interleaving ratio x (X-axis). Figure 4c shows the distribution of MLP increases on CXL: 88% of workloads exhibit increases below 10%, and 76% below 5%.
- **Reasoning:** MLP is primarily determined by the core's structural limits (*e.g.*, LFB/SQ size) and the program's data dependency, not by memory contention.

Symbol	Meaning
x	Fraction of memory footprint on DRAM ($0 \leq x \leq 1$)
x'	Fraction of memory footprint on a tier ($0 \leq x' \leq 1$)
L_{idle}	(Constant) Unloaded/idle latency for DRAM and CXL, $L_{\text{idle}}^{\text{DRAM}}$ and $L_{\text{idle}}^{\text{CXL}}$ are measured via Intel MLC [4]
L_{full}	Latency under full load, measured for each workload under DRAM ($L_{\text{full}}^{\text{DRAM}}$, $x=1$) and CXL ($L_{\text{full}}^{\text{CXL}}$, $x=0$)
$\mathcal{M}(x')$	**Load scaling factor**: Relative cycle contribution of a tier handling load fraction x'

Table 7. Notation for Camp's interleaving synthesis model.

- **Implication:** While MLP may fluctuate slightly, it does not scale proportionally with interleaving-induced latency changes. Consequently, changes in memory-active cycles are determined by the accumulation of memory latency rather than by changes in concurrency.

5.2.2 Modeling Latency Curve.

With MLP fixed, the problem reduces to modeling how latency changes with load. Accordingly, in Eq. 8 we interpret x' as the tier's effective load share induced by the interleaving ratio. As shown in Figure 11b, when memory contention occurs on each tier, latency is not static. As a tier's effective load increases (growing with its footprint fraction under weighted interleaving), latency rises slowly at first, then sharply as queues in the memory controller and interconnect approach saturation. We model this contention using a **quadratic transfer function**, capturing the empirically observed super-linear latency growth with load. This form reflects the rapid queue buildup that occurs once service capacity is approached. In contrast, without bandwidth contention ($L_{\text{full}} \approx L_{\text{idle}}$), per-tier latency remains near unloaded values and constant across interleaving ratios (Figure 11a).

$$L(x') = L_{\text{idle}} + (L_{\text{full}} - L_{\text{idle}}) \cdot x'^2 \tag{8}$$

Here, L_{idle} is the idle latency (measured at $x' \approx 0$) and L_{full} is the full-load latency (measured at $x' = 1$). When L_{full} and L_{idle} differ per tier (*i.e.*, memory contention exists), this quadratic fit closely matches empirical measurements (Figure 11b).

We do not claim that latency is universally quadratic in load. Rather, the quadratic form provides a compact and sufficiently accurate approximation over the operating range relevant to interleaving, as validated by our measurements.

5.2.3 Deriving the Load Scaling Factor.

We now define the **Load Scaling Factor**, $\mathcal{M}(x')$, which represents how many memory-active cycles a tier contributes when handling a fraction x' of the load, relative to that tier's endpoint baseline (*i.e.*, when the tier serves the entire memory footprint). Using the relation $Cycles \propto Load \times Latency$:

$$\mathcal{M}(x') = \frac{\text{Load} \times \text{Latency}}{\text{Baseline}} = \frac{x' \cdot L(x')}{1 \cdot L_{\text{full}}}$$

Substituting our quadratic latency model (Eq. 8) yields:

$$\mathcal{M}(x') = \frac{x' \cdot [L_{\text{idle}} + (L_{\text{full}} - L_{\text{idle}}) \cdot x'^2]}{L_{\text{full}}} \tag{9}$$

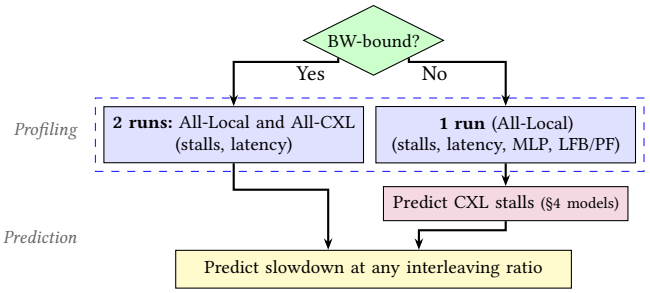

Figure 12. Workflow for performance modeling. *Bandwidth-bound workloads ($L > L_{idle}$) require two profiling runs; latency-bound workloads need only one, with CXL stalls predicted analytically.*

We compute $\mathcal{M}(x')$ separately for DRAM and CXL using each tier's measured $(L_{\text{idle}}, L_{\text{full}})$, so the scaling factor is anchored to the corresponding endpoint run.

This function captures the dominant effects of contention:

- **No contention ($L_{\text{full}} \approx L_{\text{idle}}$):** The quadratic term vanishes, and $\mathcal{M}(x') \approx x'$. Performance scales linearly.
- **High contention ($L_{\text{full}} \gg L_{\text{idle}}$):** The cubic term (x'^3) dominates. Shifting traffic *away* from this tier yields super-linear performance gains, mathematically explaining the "bathtub" curve observed in bandwidth-bound workloads.

5.2.4 The Unified Predictor.

Finally, we predict slowdown at ratio x by scaling the endpoint memory-related stall cycles from each tier using the tier-specific load factors, then normalizing by the DRAM-baseline CPU execution cycles.

$$S(x) \approx \frac{\mathcal{M}(x) \cdot s_{\text{DRAM}} + \mathcal{M}(1-x) \cdot s_{\text{CXL}} - s_{\text{DRAM}}}{c} \quad (10)$$

where s_{DRAM} and s_{CXL} denote the stall cycles for each slowdown component (s_{LLC}, s_{Cache}, and s_{SB} as defined in §4) measured when the workload runs on DRAM and CXL, respectively. We use c from the DRAM run as the normalization baseline throughout. The overall slowdown is the sum of $S_{\text{Drd}}(x)$, $S_{\text{Cache}}(x)$, and $S_{\text{Store}}(x)$.

5.3 Profiling Workflow

Figure 12 illustrates Camp's interleaving profiling strategy.

1. **Latency-bound (1 Run):** If a workload's measured memory latency in the DRAM run is close to or less than the unloaded latency ($L_{\text{idle}}^{\text{DRAM}}$), it is not bandwidth-bound. We classify a workload as latency-bound if its measured DRAM latency is within τ of $L_{\text{idle}}^{\text{DRAM}}$, where τ is a small platform-specific tolerance (*e.g.*, 5%). The model simplifies to linear interpolation ($L(x')$ is constant). Because we can predict CXL stalls from DRAM (§4), we need *only one DRAM run.*
2. **Bandwidth-bound (2 Runs):** If latency is elevated, contention exists. We use a 2nd run on CXL to measure the endpoint s_{CXL} and then rely on Eq. 10 to synthesize the full interleaving performance curve.

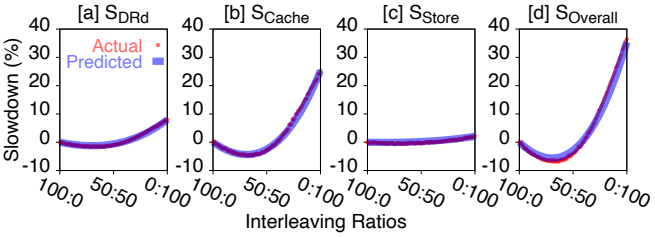

Figure 13. Prediction accuracy. *The actual vs. predicted slowdown for 10-thread* `603.bwaves` *under interleaving ratios from 100:0 to 0:100. (a)–(c) show results for per-component slowdown (S_{DRd}, S_{Cache}, and S_{Store}). (d) shows the sum of all sub-slowdowns.*

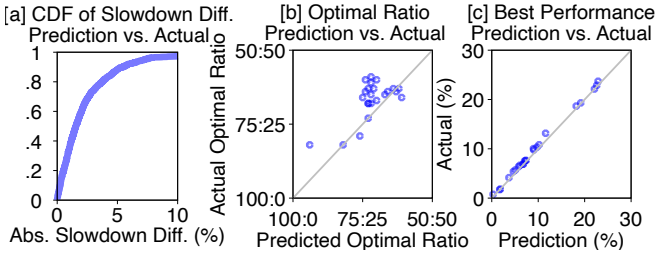

Figure 14. Accuracy of interleaving prediction. *(a) CDF of mispredictions; (b) Predicted versus actual optimal interleaving ratio; (c) Best interleaving performance under Best-shot vs. oracle.*

5.4 Interleaving Model Evaluation

Accuracy. Figure 13 visualizes the prediction for `603.bwaves`.

The model reconstructs the convex performance curve, closely matching the actual measured slowdowns across all 99 ratios (from 99:1 to 1:99). Across 20 bandwidth-bound workloads from SPEC CPU 2017 and Llama, **90% of predictions fall within 5% absolute slowdown error** (Figure 14a).

Finding the optimum. Figure 14b compares the predicted *vs.* actual optimal interleaving ratios. While the model is slightly conservative, Figure 14c confirms that the **performance achieved** by the predicted ratios is practically identical to the oracle optimum. This confirms that Camp can effectively guide "Best-Shot" interleaving policies (§6.1), jumping directly to the optimal configuration without iterative search.

5.5 Limitations and Future Work

Camp's interleaving model currently applies to weighted interleaving policies. An important future direction is to extend this framework to first-touch-based allocation and migration-driven tiered memory management, enabling performance synthesis under dynamic page placement policies. Additionally, extending the model to capture cross-tier interference effects and non-uniform workload phases could further improve accuracy.

6 Camp Use Cases

This section demonstrates how Camp's predictive models translate into practical system-level benefits. We focus on

two representative use cases: (1) Best-shot, which selects a near-optimal interleaving ratio without online search, and (2) colocated workload placement across memory tiers. In both cases, Camp outperforms state-of-the-art policies by reasoning directly about performance slowdown, rather than relying on indirect proxies such as latency or miss counts.

6.1 Best-shot Interleaving Policy

Best-shot is an interleaving policy derived from Camp's synthesized interleaving performance model. For a given workload, Best-shot leverages the predicted performance curve to provide three key capabilities. First, it determines whether a workload can benefit from aggregate DRAM and CXL bandwidth, regardless of whether the workload is latency-sensitive or bandwidth-bound. Second, it predicts the interleaving ratio that minimizes execution time. Third, it accurately forecasts performance at arbitrary interleaving ratios. Importantly, Best-shot also handles workloads that do not benefit from CXL bandwidth. In such cases, it predicts the interleaving configuration that minimizes slowdown relative to DRAM-only execution, allowing users to proactively avoid harmful configurations.

6.2 "Best-shot" vs. Existing Tiering Policies

6.2.1 Experimental setup. We evaluate Best-shot on eight bandwidth-bound workloads from SPEC CPU 2017 and Llama. Across the evaluated workloads, Best-shot uses only 62–74% of the fast-tier capacity. To avoid disadvantaging the baseline systems, we provision them with a fixed 4:1 fast-to-slow tier ratio (*i.e.*, 80% fast memory).

We compare Best-shot against seven baselines: **(1)** "Interleave 1-1", Linux default 1:1 interleaving policy (`MPOL_INTERLEAVED`), **(2)** Caption [46], which searches over a coarse set of interleaving ratios using latency-based heuristics, **(3)** First-touch placement without proactive migrations, **(4)** NUMA Balancing Tiering (NBT) [7, 8], memory tiering support in recent Linux, **(5)** Colloid [51], a tiering system that aims to equalize access latency across tiers, **(6)** Alto [38], a tiering policy (on top of Colloid) that limits page migration during high-MLP time intervals, and **(7)** Soar [38], a profile-guided allocation policy that places performance-critical objects on DRAM.

6.2.2 Overall results. Figure 15 shows that Best-shot consistently outperforms all baseline approaches across the evaluated workloads, achieving up to 21% performance improvement over First-touch and substantial gains over Caption, NBT, Colloid, Alto, and Soar. While Caption improves upon static policies, its effectiveness is limited by a coarse-grained search space. Migration-based tiering policies (NBT, Colloid, and Alto) are effective in latency-constrained scenarios. However, under high bandwidth pressure, they fail to fully exploit the aggregate DRAM and CXL bandwidth, often underperforming the DRAM-only configuration and incurring nontrivial migration overheads. Similarly, Soar's strategy of placing most performance-critical objects in the fast-tier underutilizes CXL bandwidth and consequently underperforms interleaving-based approaches.

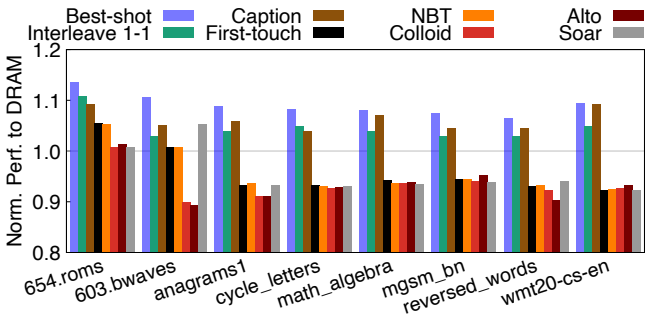

Figure 15. Best-shot vs. others. *Best-shot exploits aggregate DRAM and CXL bandwidth and consistently outperforms 7 baseline policies across 8 bandwidth-bound workloads. Performance is normalized to DRAM-only execution. Y-axis starts at 0.8. Higher is better.*

6.2.3 Understanding Best-shot performance gains. We now analyze why Best-shot outperforms existing approaches and identify the sources of its performance advantages.

Best-shot vs. Colloid and NBT. Colloid is designed around the principle of latency equivalence, migrating pages to equalize access latency across tiers. However, equalizing latency does not imply optimal performance. Consider `654.roms`: under Colloid, the average DRAM and CXL latencies are approximately 168ns (1.9× of idle DRAM latency) and 189ns (2.1×), respectively. To reduce the gap, Colloid aggressively migrates pages into DRAM, increasing contention on DRAM.

In contrast, Best-shot predicts an interleaving ratio that reduces DRAM contention, lowering DRAM latency to 139ns while CXL latency is 191ns. The reduced contention on local memory yields 13% higher performance than Colloid. Figure 11 confirms this behavior: at the optimal point, DRAM latency can be lower than CXL latency, and attempts to equalize them degrade performance. This directly contradicts the core assumption underlying latency-equalization policies. While latency is an important indicator, stall cycles are a more direct performance proxy. Accordingly, Camp models how latency translates into stall cycles and overall execution time. NBT relies on recency-based hotness rather than explicit latency equalization, making it less aggressive in migration under contention. Consequently, it outperforms Colloid on several workloads but remains consistently inferior to Best-shot.

Best-shot vs. Caption. Caption determines the interleaving ratio by probing a small set of coarse-grained configurations. In contrast, Best-shot analytically synthesizes the entire performance curve and predicts the optimal ratio with fine granularity. As shown in Figure 14b, many optimal interleaving ratios are below 80% fast-tier memory usage. First-touch cannot exploit this property, while search-based methods incur substantial profiling overhead.

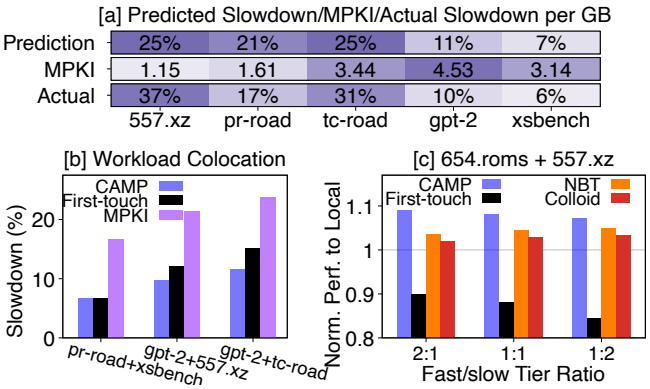

Figure 16. Camp-guided workload colocation. *(a) shows Camp accurately predicts slowdown under colocation; (b) shows that colocation guided by MPKI worsens the performance; (c) shows Camp provides better placement than other policies for colocated workloads.*

Best-shot vs. Alto and Soar. Alto reduces page migration during high-MLP intervals but fails to leverage aggregate memory bandwidth effectively, so Alto is slightly better than Colloid. Soar places performance-critical objects on DRAM, causing high memory contention, *e.g.*, `654.roms` experiences 13% worse performance than Best-shot.

Takeaway #4: Best-shot demonstrates that analytically predicting interleaving performance enables simple yet effective tiering decisions, delivering superior performance compared to state-of-the-art tiering and interleaving strategies in bandwidth-bound scenarios.

6.3 Colocated Workload Scheduling

Motivation and setup. We next evaluate Camp as a predictor for colocated workload scheduling. When multiple workloads share heterogeneous memory, inaccurate performance proxies can lead to poor placement decisions. We select three pairs of latency-bound workloads where Camp and MPKI, a commonly used cache-miss metric, predict conflicting slowdown rankings. Each workload pair is colocated under a 1:1 fast-to-slow tier configuration.

Prediction accuracy vs. MPKI. Figure 16a compares Camp-predicted slowdown, MPKI-based heuristics, and the actual measured slowdown during colocation. Across all cases, Camp predictions closely track observed slowdown, while MPKI provides misleading signals. For example, `gpt-2` exhibits low MPKI but high slowdown under CXL, whereas `tc-road` shows high MPKI but relatively low slowdown. MPKI fails to capture tolerance to latency and interference.

Impact on placement decisions. Figure 16b shows that MPKI-guided placement leads to 10%, 11.6%, and 12.2% worse performance than Camp-guided placement across the evaluated workload pairs. This is because MPKI does not reflect how memory latency translates into stall cycles, whereas Camp explicitly models this effect, even under contention.

Mixed bandwidth-bound and latency-bound colocation. Figure 16c considers colocating a bandwidth-bound workload, `654.roms`, with a latency-bound workload, `557.xz`, under different fast-to-slow tier ratios. Best-shot assigns `654.roms` its predicted optimal interleaving ratio and places `557.xz` in the remaining fast memory. This configuration outperforms First-touch, NBT, and Colloid across all ratios by simultaneously exploiting aggregate bandwidth and protecting latency-sensitive execution.

Takeaway #5: Colocation results highlight that accurate slowdown prediction is critical for multi-workload memory management. By reasoning about performance impact instead of cache misses or raw latency, Camp enables placement decisions that improve both individual and system-level performance.

6.4 Discussion

While this study focuses on CXL, the high prediction accuracy on NUMA indicates a pathway to performance observability for general memory systems using lightweight performance counters. Our models are validated across three CXL devices and NUMA, demonstrating broad applicability. The simplicity of Camp models facilitates both offline capacity planning and resource management. We envision more use cases of Camp in guiding hybrid memory policies that integrate interleaving and tiering, as well as improved profiling for tiered memory systems [27].

7 Conclusion

Camp demonstrates that CXL slowdown can be accurately predicted without running on CXL, by grounding models in causal microarchitectural mechanisms rather than correlation-based heuristics. This principle extends beyond the specific predictors we present: as memory hierarchies grow more heterogeneous, understanding *why* performance degrades, not just *that* it does, becomes essential for principled system design. Our evaluation shows that these models enable practical policies, Best-shot interleaving and interference-aware colocation, that outperform existing approaches by up to 21% and 23%, respectively. We hope Camp inspires further work on interpretable, hardware-grounded models that enable proactive management of emerging memory technologies.

Acknowledgments

We thank Boris Grot (our shepherd) and the anonymous reviewers for their constructive feedback. We also thank CloudLab for providing the infrastructure used in our experimental evaluation. This research was partially supported by the NSF CAREER Award CNS-2339901, NSF Grant CNS-2312785, Google, and Microsoft. Jinshu Liu is supported by a Google PhD Fellowship.

References

[1] ARM11 MPCore Processor Technical Reference Manual r2p0. https://developer.arm.com/documentation/ddi0360/f/level-1-memory-system/about-the-level-1-data-side-memory-system/linefill-buffers.

[2] Deep Learning Recommendation Model (DLRM). https://github.com/facebookresearch/dlrm.

[3] GPT-2. https://github.com/openai/gpt-2.

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

[5] Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1. https://cdrdv2-public.intel.com/671488/248966-Software-Optimization-Manual-V1-048.pdf.

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

[7] Memory Tiering: Hot Page Selection. https://lwn.net/Articles/898615/.

[8] mm/demotion: Memory Tiers and Demotion. https://lwn.net/Articles/897026/.

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

[10] Phoronix. https://github.com/phoronix-test-suite/phoronix-test-suite.

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

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

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

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

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

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

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

[18] 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.

[19] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[20] Chiachen Chou, Aamer Jaleel, , and Moinuddin Qureshi. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM. In *The International Symposium on Memory Systems (MEMSYS)*, 2017.

[21] Yuan Chou, Brian Fahs, and Santosh Abraham. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[22] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *Proceedings of the 28th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2019.

[23] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, 2016.

[24] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[25] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A Mechanistic Performance Model for Superscalar Out-of-Order Processors. *ACM Transactions on Computer Systems*, 27(2), 2009.

[26] Björn Gottschall, Lieven Eeckhout, and Magnus Jahre. TEA: Time-Proportional Event Analysis. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[27] Hamid Hadian, Jinshu Liu, Hanchen Xu, Hansen Idden, and Huaicheng Li. PACT: A Criticality-First Design for Tiered Memory. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2026.

[28] 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.

[29] Wentao Huang, Mian Lu, and Kian-Lee Tan. Hash Joins Meet CXL: A Fresh Look. In *Proceedings of the 18th Conference on Innovative Data Systems Research (CIDR)*, 2026.

[30] 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.

[31] Georgiy Lebedev, Hamish Nicholson, Musa Ünal, Sanidhya Kashyap, and Anastasia Ailamaki. Demystifying CXL Memory Bandwidth Expansion for Analytical Workloads. In *International Workshop on Accelerating Analytics and Data Management Systems (ADMS)*, 2025.

[32] Hwanjun Lee, Minho Kim, Yeji Jung, Seonmu Oh, Ki-Dong Kang, Seunghak Lee, and Daehoon Kim. Beyond Page Migration: Enhancing Tiered Memory Performance via Integrated Last-Level Cache Management and Page Migration. In *58th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-58)*, 2025.

[33] 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.

[34] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[35] Xiao Li, Zerui Guo, Yuebin Bai, Mahesh Ketkar, Hugh Willkinson, and Ming Liu. Understanding and Profiling CXL.mem Using PathFinder. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2025.

[36] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S. Berger, Marie Nguyen, Xun Jian, Sam H. Noh, and Huaicheng Li. Systematic CXL Memory Characterization and Performance Analysis at Scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

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

[38] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. Tiered Memory Management Beyond Hotness. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2025.

[39] Zhihong Luo, Sam Son, Sylvia Ratnasamy, and Scott Shenker. Harvesting Memory-bound CPU Stall Cycles in Software with MSH. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[40] Yirong Lv, Bin Sun, Qinyi Luo, Jing Wang, Zhibin Yu, and Xuehai Qian. CounterMiner: Mining Big Performance Data from Hardware Counters. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*, 2018.

[41] Fabian Mahling, Marcel Weisgut, and Tilmann Rabl. Fetch Me If You Can: Evaluating CPU Cache Prefetching and Its Reliability on High Latency Memory. In *21st International Workshop on Data Management on New Hardware (DaMoN)*, 2025.

[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] Sanyam Mehta. Performance Analysis and Optimization with Little's Law. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022.

[44] Zhenlin Qi, Shengan Zheng, Ying Huang, Yifeng Hui, Bowen Zhang, Linpeng Huang, and Hong Mei. Chrono: Meticulous Hotness Measurement and Flexible Page Migration for Memory Tiering. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys)*, 2025.

[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] 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.

[47] 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 19th European Conference on Computer Systems (EuroSys)*, 2024.

[48] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, 2014.

[49] Musa Unal, Vishal Gupta, Yueyang Pan, Yujie Ren, and Sanidhya Kashyap. Tolerate It if You Cannot Reduce It: Handling Latency in Tiered Memory. In *Proceedings of the 20th Workshop on Hot Topics in Operating Systems (HotOS)*, 2025.

[50] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. In *IEEE Symposium on Security and Privacy (SP)*, 2021.

[51] Midhul Vuppalapati and Rachit Agarwal. Tiered Memory Management: Access Latency is the Key! In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

[52] Midhul Vuppalapati, Saksham Agarwal, Henry Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. Understanding the Host Network. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2024.

[53] Xi Wang, Jie Liu, Jianbo Wu, Shuangyan Yang, Jie Ren, Bhanu Shankar, and Dong Li. Performance Characterization of CXL Memory and Its Use Cases. In *Proceedings of the 39th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025.

[54] Marcel Weisgut, Daniel Ritter, Pınar Tözün, Lawrence Benson, and Tilmann Rabl. CXL Memory Performance for In-Memory Data Processing. In *Proceedings of the 51st International Conference on Very Large Data Bases (VLDB)*, 2025.

[55] 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.

[56] 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.

[57] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. MT^2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In *Proceedings of the 20th USENIX Symposium on File and Storage Technologies (FAST)*, 2022.

[58] Xinyue Yi, Hongchao Du, Yu Wang, Jie Zhang, Qiao Li, and Chun Jason Xue. ArtMem: Adaptive Migration in Reinforcement Learning-Enabled Tiered Memory. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025.

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