CAPO: Enhancing CXL-based Memory Expanders with Adaptive Prefetching

Jingyan Song
National Innovation Institute of Defense Technology
Academy of Military Sciences
Beijing, China
jingyan.song@hotmail.com

Chenglong Li
National Innovation Institute of Defense Technology
Academy of Military Sciences
Beijing, China
chenglongli17@163.com

Anwen Huang*
National Innovation Institute of Defense Technology
Academy of Military Sciences
Beijing, China
awhuang@163.com

Qiong Li
National Innovation Institute of Defense Technology
Academy of Military Sciences
Beijing, China
517066092@qq.com

Yongji Liu
National Innovation Institute of Defense Technology
Academy of Military Sciences
Beijing, China
liuyongji77@163.com

Zhijie Liu
National Innovation Institute of Defense Technology
Academy of Military Sciences
Beijing, China
1498954197@qq.com

Na Nie
National Innovation Institute of Defense Technology
Academy of Military Sciences
Beijing, China
15108621614@163.com

Abstract

Compute Express Link (CXL) has emerged as a key technology for memory expansion. Bandwidth and latency are the two critical metrics for memory devices, and CXL memory exhibits 2–3× higher access latency than local DDR DRAM, degrading application performance. Device-side prefetching, integrated inside the CXL controller and transparent to applications, can hide this latency. However, we find that existing device-side prefetching encounters two challenges under high concurrency. (1) The CXL controller lacks coordinated demand–prefetch tracking, so overlapping demand and prefetch accesses generate redundant backend reads. (2) Existing prefetchers lack load-aware degree adaptation, so prefetch traffic crowds out demand bandwidth under high load. We propose CAPO (CXL Adaptive Prefetch cOntroller), an adaptive device-side prefetch controller for CXL-based memory expanders. CAPO has two components. CapoMSHR tracks inflight demand and prefetch requests separately and eliminates duplicate backend reads when a demand hits an inflight prefetch. CapoTuner uses online reinforcement learning to set the prefetch degree each window from device-observable signals, tracking an appropriate operating point across workloads and concurrency levels. We evaluate CAPO on seven memory-intensive workloads with diverse access behaviors at 1–32 cores. CAPO achieves a 3.1%–13.3% geometric-mean speedup over the no-prefetch baseline. At 16 and 32 cores, CAPO turns the −7.0% and −13.3% geometric-mean slowdowns of fixed-degree static prefetching into +10.3% and +3.1% speedups.

CCS Concepts: • **Computer systems organization** → **Interconnection architectures**; • **Hardware** → *Buses and high-speed links*; • **Computing methodologies** → *Reinforcement learning*.

Keywords: Compute Express Link, memory expansion, device-side prefetching, prefetch throttling, reinforcement learning

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-XXXX-XXXX-X/26/10
https://doi.org/10.1145/XXXXXXX.XXXXXXX

1 INTRODUCTION

Memory-intensive workloads [15, 27, 37] continue to grow in working set size, placing increasing demands on memory bandwidth and capacity. Scaling single-node memory is increasingly difficult due to the physical constraints of the DDR

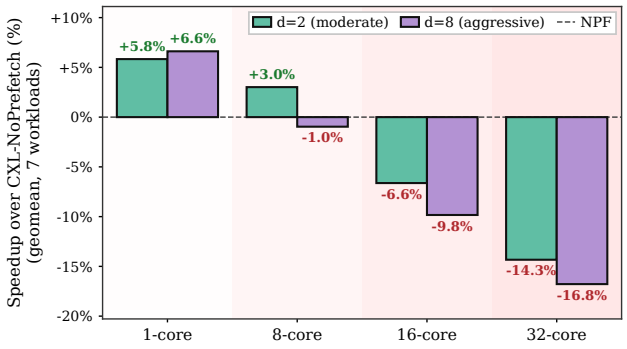


Figure 1. Static Prefetching Breaks Down at High Concurrency (d denotes prefetch degree).

interface [27, 34]. Each memory channel requires hundreds of signal pins, and the limited CPU package and PCB area cap the number of accessible channels. Compute Express Link (CXL) [9] enables CPUs to access CXL-based memory expanders (CXL memory) over the PCIe physical layer with unified load/store semantics and hardware cache coherence. Compared with remote-memory alternatives such as RDMA [1, 14, 28], CXL memory typically offers lower access latency and has been deployed in production systems [21, 24, 40]. Nevertheless, measured CXL memory access latency typically falls in the range of 200–400 ns, 2–3× that of local DRAM [21, 23, 24, 42], imposing a meaningful performance penalty on memory-intensive workloads.

Data prefetching is a classic latency-hiding technique [2, 26]. A prefetch mechanism speculatively brings data into a storage structure closer to the requester, such as a cache or prefetch buffer, before demand accesses arrive, reducing the impact of high access latency on application performance. In CXL memory, device-side prefetching places a prefetch unit inside the CXL controller, observes host memory traffic at the CXL link egress without modifying host software or CPU microarchitecture, and remains transparent to upper layers. Polaris [44] systematically evaluates device-side prefetching for CXL memory and demonstrates its latency-hiding potential. In CXL-SSD scenarios, LG-Prefetcher [22] and other CXL-SSD system designs [43] explore similar device-side prefetching ideas.

Existing device-side prefetch designs leave two issues unresolved. (1) Device-side prefetching for CXL memory lacks demand–prefetch coordination. A later demand can trigger another backend read to an address that already has an inflight prefetch, amplifying backend traffic and memory pressure. (2) Existing designs also neglect link bandwidth pressure in degree control, either using a fixed static degree or adjusting degree solely from accuracy and coverage metrics [22, 43, 44], so prefetch traffic crowds out demand bandwidth under high load.

We systematically analyze static device-side prefetching as represented by Polaris and find that its benefit erodes as core count scales. Across prefetch degrees $d \in \{1, 2, 4, 8\}$, static prefetching drops to a geometric-mean slowdown of −7.0% at 16 cores and −13.3% at 32 cores. The same erosion appears at both moderate degree (d=2) and aggressive degree (d=8) settings (Figure 1). We attribute the collapse to load-sensitive latency inflation, prefetch accuracy degradation and bandwidth pressure in aggregated multi-core streams, and missing demand–prefetch coordination on the device side. We detail each failure mode in Section 3.

There are two core challenges that remain unresolved in existing prefetching schemes. (1) A CXL controller must eliminate redundant backend reads caused by demand–prefetch overlap, namely the duplicate read triggered when a later demand overlaps an inflight prefetch to the same address. (2) Prefetch degree must match link bandwidth pressure and suppress over-prefetching under high load. Otherwise, prefetch traffic crowds out demand bandwidth and erases or reverses prefetch gains.

We propose **CAPO** (CXL Adaptive Prefetch cOntroller), an adaptive device-side prefetch controller deployed inside the CXL controller and fully transparent to the host. CAPO has two components. **CapoMSHR** addresses Challenge 1 by tracking inflight demand and prefetch requests separately and coordinating them at the address level. When a demand hits an already-issued prefetch, the demand attaches to the prefetch and reuses its response. When a demand or write hits a queued prefetch candidate, CapoMSHR cancels the candidate to avoid waste. CapoMSHR substantially reduces redundant backend reads with minimal hardware overhead. **CapoTuner** addresses Challenge 2 with a per-window online learning controller that incorporates link load into its reward, suppressing over-prefetching under high concurrency while adapting to workload shifts without offline profiling.

We make the following contributions.

- We characterize the performance collapse of static device-side prefetching on CXL memory at 1, 8, 16, and 32 cores, and identify three failure modes (1) load-sensitive latency inflation, (2) accuracy and bandwidth degradation in aggregated multi-core streams, and (3) redundant backend reads from missing device-side demand–prefetch coordination.
- We propose CapoMSHR, a device-side structure that separately tracks inflight demand and prefetch requests and merges their overlap into a single backend read.
- We design CapoTuner, an online learning controller that folds link bandwidth pressure into its reward and adapts prefetch degree per window to suppress over-prefetching under high concurrency.
- We evaluate CAPO in SimCXL [41, 42] on seven memory-intensive workloads at 1, 8, 16, and 32 cores,

achieving a 3.1%–13.3% geometric-mean speedup over the no-prefetch baseline and converting the −7.0% and −13.3% slowdowns of fixed-degree static prefetching at 16 and 32 cores into +10.3% and +3.1% speedups.

2 BACKGROUND

2.1 CXL Memory

Compute Express Link (CXL) [9] is an open interconnect standard built on the PCIe 5.0/6.0 physical layer. CXL targets high-bandwidth, low-latency connectivity between CPUs, accelerators, and memory expanders. The CXL protocol suite comprises three sub-protocols. **CXL.io** handles device discovery, configuration, and I/O initialization, analogous to standard PCIe. **CXL.cache** allows attached accelerators (e.g., GPUs, FPGAs) to coherently cache host memory. **CXL.mem** enables the host CPU to access device-attached memory via load/store instructions with hardware cache coherence.

CXL devices are classified by sub-protocol support. Type 1 supports CXL.io and CXL.cache, Type 2 supports all three sub-protocols, and Type 3 supports CXL.io and CXL.mem. CAPO targets CXL Type 3 devices, which are dedicated memory expanders that extend host-addressable capacity without changes to host software.

Despite hardware cache coherence and load/store semantics, CXL protocol processing overhead and PCIe link traversal add latency over local DRAM, limiting performance for memory-intensive workloads.

2.2 Device-Side Prefetching

Device-side prefetching embeds a prefetch unit inside the CXL memory controller. The prefetch unit observes the host request stream at the CXL link egress, infers future accesses from observed patterns, and proactively fetches anticipated data into a dedicated on-device SRAM prefetch buffer ahead of demand.

Bandwidth headroom. Under CXL 1.1/2.0, encoding overhead and Flit-level packaging cap the effective read bandwidth at roughly 80%–90% of the physical-layer peak [10, 23, 30, 35, 42]. The backend DRAM of a Type 3 expander typically sustains higher peak bandwidth than the unidirectional CXL link budget, so a device-side prefetch unit can use the residual backend bandwidth to stage data into the on-device SRAM buffer without contending with demand traffic on the link.

No host-side cost. Prefetching avoids two host-side costs. (1) Prefetched data resides in the device-side buffer rather than the host last-level cache, so prefetching does not consume LLC capacity. (2) Device-side prefetching requires no modifications to the host CPU microarchitecture, OS, or applications.

Lower effective latency. A demand request that hits the prefetch buffer is served directly from on-device SRAM and bypasses the backend DRAM access, substantially reducing observed access latency.

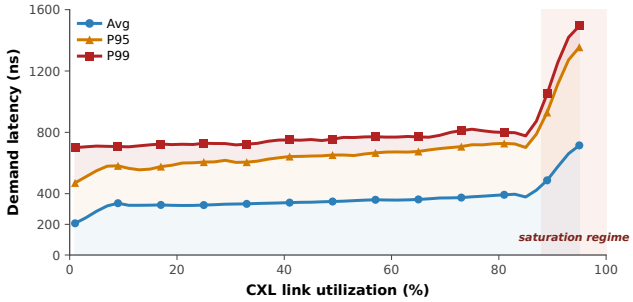


Figure 2. CXL Memory Latency Grows Nonlinearly with Link Utilization.

Polaris demonstrates the latency-hiding potential of device-side prefetching for CXL memory [44]. CXL-SSD systems [22, 43] explore similar ideas for storage devices.

2.3 Reinforcement Learning for Adaptive Control

Reinforcement learning (RL) [17, 36] learns a decision policy through repeated interaction with an environment. RL is well-suited to control problems where exploratory actions carry real cost. SARSA [36] is an on-policy, value-based RL algorithm whose updates evaluate the policy under which actions are actually executed. The on-policy update yields conservative behavior in dynamic environments. RL-based controllers have been applied to online prefetch degree adaptation [7, 8] and memory controller optimization [17].

CapoTuner adopts SARSA because an aggressive probe on a production CXL controller can directly degrade demand performance under high load. SARSA's on-policy update lets the controller learn from the consequences of its own (potentially conservative) exploration rather than from a hypothetical greedy baseline.

3 MOTIVATION

CAPO is motivated by three observations about CXL device-side prefetching under load. Together, the three observations show that effective control must reason about both system pressure and demand–prefetch interaction.

O1. CXL memory exhibits strong load-sensitive latency inflation. Queuing theory establishes that memory access latency grows non-linearly with bandwidth utilization [18]. At low utilization, latency stays near the unloaded baseline. As utilization approaches service capacity, queuing effects amplify both average and tail latency. Local DRAM systems mitigate this effect via multi-channel organizations and host-side memory scheduling, but real-system measurements confirm that CXL memory remains susceptible because of its bandwidth-constrained remote-memory operating regime [23]. Our simulation reproduces the same trend

on a representative CXL workload (XSBench, Figure 2). The P99 tail latency reaches several times the unloaded baseline once link utilization crosses 88%, while average latency grows much more slowly.

O2. Static device-side prefetching turns from beneficial to harmful under multi-core contention. The effectiveness of device-side prefetching depends critically on access stream regularity and system load. At low core counts the access stream observed by the device is relatively structured, letting the prefetch unit capture patterns accurately and fetch data ahead of demand. As core count scales, the situation changes. Static degree-2 prefetching delivers geometric-mean IPC improvements of +5.8% at 1 core and +3.0% at 8 cores, but declines to −6.6% at 16 cores and −14.3% at 32 cores. The more aggressive degree-8 setting follows the same trend, with +6.6%, −1.0%, −9.8%, and −16.8% at 1, 8, 16, and 32 cores respectively (Figure 1).

Two factors drive the collapse. (1) After the LLC, requests from multiple cores merge into a heavily multiplexed aggregate stream with substantially lower regularity than the single-core stream, degrading the prefetch unit's pattern-recognition accuracy. (2) Multi-core concurrency elevates load on both the CXL link and the device backend. Combined with the load-sensitive characteristics of O1, the additional traffic injected by prefetching under high load exacerbates queuing delays for demand requests instead of hiding latency, turning prefetching into a source of performance degradation.

O3. The absence of coordinated request tracking causes redundant backend reads. In conventional CPU memory hierarchies, the LLC employs MSHRs [12, 20] to merge inflight miss requests to the same address, preventing duplicate requests from reaching lower memory levels. CXL Type 3 device controllers typically assume that such deduplication has already been handled by the host and therefore do not provision equivalent request-tracking structures on the device side.

Device-side prefetch requests and demand requests originate from two independent paths, and the LLC's MSHRs have no visibility into prefetches autonomously issued by the device. A completed prefetch can still serve a later demand directly from the prefetch buffer (Figure 3a ①). The problematic cases arise when the prefetch and the demand overlap in time. (1) A prefetch already issued to the backend overlaps a later demand for the same address that misses the LLC. Without inflight prefetch tracking on the device, the demand triggers a second backend read for the same line (Figure 3a ②). (2) A queued prefetch candidate overlaps a demand for the same address. The candidate's queue slot becomes wasted, and its eventual issue competes with the demand for backend bandwidth. Under high load, the redundant traffic compounds the load-sensitive effects of O1 and O2, pushing the system more quickly into rapid latency degradation.

The three observations expose two core challenges in CXL device-side prefetch control.

Challenge 1 (from O3). Elimination of redundant backend reads caused by demand–prefetch overlap. A CXL controller must avoid issuing a duplicate backend read when a later demand overlaps an inflight prefetch to the same address.

Challenge 2 (from O1, O2). Prefetch degree adaptation under link bandwidth pressure. Prefetch degree must match link bandwidth pressure under high load. Otherwise, over-prefetching crowds out demand bandwidth and erases or reverses prefetch gains.

4 CAPO DESIGN AND IMPLEMENTATION

4.1 Overview

CAPO is a device-side prefetch control framework for CXL Type 3 memory-expansion devices. CAPO sits entirely inside the CXL controller, between the CXL.mem interface and the backend memory, and stays transparent to the CPU, the cache-coherence protocol, and system software. Two new components address the challenges raised in Section 3.

CapoMSHR (guided by O3). CapoMSHR separately tracks the inflight state of demand and prefetch requests on the device side and coordinates the two paths, removing the redundant backend reads caused by overlapping accesses (Challenge 1).

CapoTuner (guided by O1, O2). CapoTuner tunes the prefetch degree online from device-visible signals of bandwidth, latency, and prefetch effectiveness, suppressing over-prefetching at high load while remaining adaptive to workload shifts (Challenge 2).

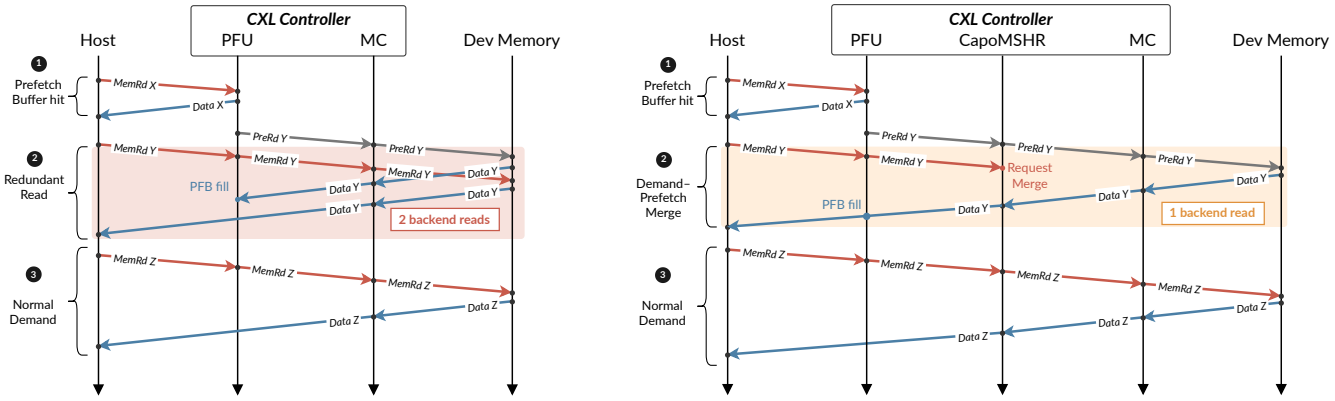
CAPO organizes these components around three controller-side paths (Figure 4).

Demand path. Demand requests from the host enter CAPO through the CXL.mem interface. A hit on the on-device prefetch buffer is served directly. On a miss, CapoMSHR probes the P-MSHR. A P-MSHR hit attaches the demand to the inflight prefetch. A miss allocates a D-MSHR entry, queues in the demand queue, and reaches the backend through the dual-queue scheduler.

Prefetch path. The Polaris-style ensemble prefetch unit [4, 19, 25, 32, 44] produces candidates that CapoMSHR holds in a candidate table until issue. Issued prefetches enter the P-MSHR and the prefetch queue, then reach the backend after the demand queue drains. Returned data lands in the SRAM prefetch buffer for use by subsequent demands.

Control. CapoTuner sits in the device monitor, samples device-visible signals each window, and updates the prefetch degree fed back into the prefetch unit.

Section 4.2 details CapoMSHR. Section 4.3 details CapoTuner.



(a) Baseline. An overlapping demand issues a redundant read.

(b) CapoMSHR. The demand attaches to the inflight prefetch.

Figure 3. CapoMSHR Eliminates Redundant Backend Reads.

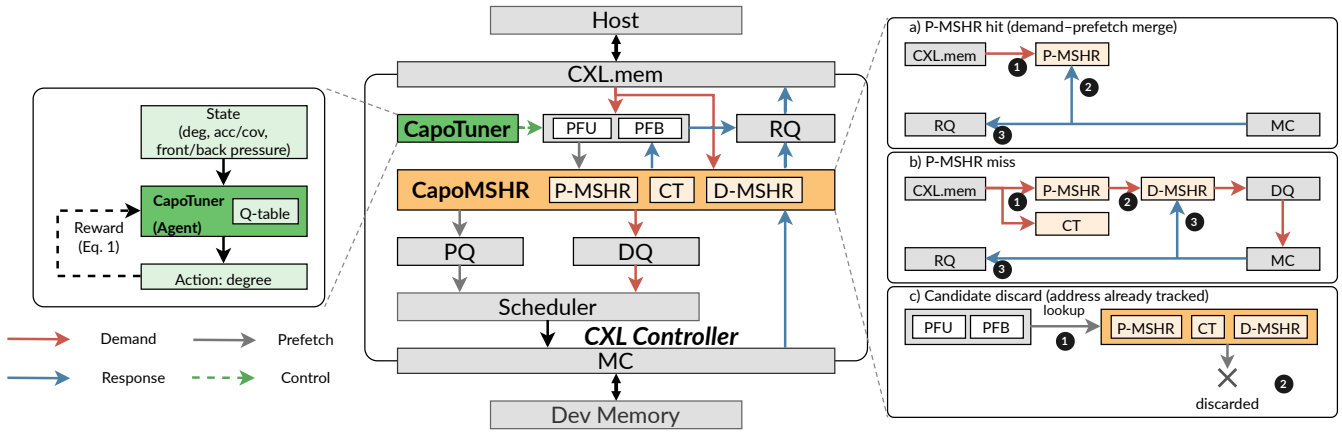


Figure 4. Architecture of CAPO. CapoMSHR (orange) and CapoTuner (green) are the two added components; the insets expand CapoMSHR's three demand–prefetch outcomes. PFU (prefetcher) and PFB (prefetch buffer) together form the prefetch unit. PQ/DQ/RQ: prefetch/demand/response queues. CT: candidate table.

4.2 CapoMSHR Design

Guided by O3, CapoMSHR tracks demand and prefetch requests in separate MSHRs [39] rather than sharing a unified MSHR between the two request types. Demand requests reaching the CXL device have already passed through the host LLC, where concurrent misses to the same cache line are merged by LLC MSHRs. The device therefore does not repeat demand–demand merging. Instead, CapoMSHR coordinates overlaps between host demand requests and device-side prefetches. The split MSHR organization also prevents device-side prefetch entries from crowding out demand entries at high prefetch degrees.

Structure. CapoMSHR has three structures. (1) The D-MSHR tracks inflight demand requests sent to the backend. (2) The P-MSHR tracks issued prefetches still in flight. (3) The candidate table holds queued prefetches that the prefetch unit has produced but has not yet issued. Before allocation, candidate generation checks the D-MSHR, P-MSHR, candidate table, and prefetch buffer (PFB). A candidate is discarded when its address already appears in any of them. The discard path is shown in Figure 4(c). The P-MSHR and the candidate table together describe the full state of the prefetch path. Inflight prefetches in the P-MSHR can be reused by a later demand. Unissued prefetches in the candidate table are cancellable speculative state.

Mechanism boundary. Within CapoMSHR, issued prefetches and queued prefetches receive asymmetric treatment. A demand attaches only to an already-issued prefetch, because an issued prefetch has a committed backend access and return path. A demand never waits on a queued prefetch, because a queued prefetch has not claimed backend resources and has no guaranteed completion time.

The asymmetric treatment removes redundant backend reads when an inflight prefetch overlaps a demand, while leaving the demand free to follow the normal miss path otherwise.

Demand–prefetch coordination on reads. A demand read first checks the prefetch buffer. On a buffer hit, the demand is served immediately. On a buffer miss, CapoMSHR probes the P-MSHR and the candidate table in parallel. (1) On a P-MSHR hit, the line is already being fetched, so the demand attaches to the inflight prefetch and reuses its response (Figure 4(a)). CapoMSHR replaces the baseline's redundant backend read (Figure 3a ②) with a single backend access (Figure 3b ②). (2) On a P-MSHR miss, the demand follows the normal miss path through the D-MSHR and demand queue (Figure 4(b)). If the candidate table also holds a queued prefetch for the same address, CapoMSHR cancels it. A queued prefetch has uncertain completion time and must not block the critical path.

Demand–prefetch coordination on writes. Writes always propagate to the backend through the demand queue. CapoMSHR then removes stale prefetch state for the same address: (1) cancel any queued candidate, (2) mark any inflight prefetch response invalid and discard it on return, and (3) invalidate the matching prefetch-buffer entry.

4.3 CapoTuner Design

Observations O1 and O2 show that an appropriate prefetch degree depends on both prefetch effectiveness and system pressure and varies across workloads and concurrency levels. CapoTuner therefore adjusts the prefetch degree online at the CXL device, one window at a time. The CXL device observes only the aggregate request stream arriving from the host LLC. CapoTuner groups a fixed number of demand reads into an observation window and uses device-side statistics to characterize the window's operating state. A prefetch degree selected at one window boundary is applied during the next window. When that window ends, CapoTuner encodes the current state from the statistics accumulated within the window, computes the reward for the applied action, updates the state–action value $Q(s, a)$ (hereafter, the Q-value) with SARSA [36] (motivation in Section 2.3), and selects a new prefetch degree for the next window.

State. The CapoTuner state encodes both prefetch benefit and system pressure, using only signals that can be measured reliably on the device side. Benefit signals include the current prefetch degree, prefetch accuracy, and prefetch coverage, characterizing the controller's operating point and the actual contribution of prefetching to demand accesses. Pressure signals include frontend read pressure and backend bandwidth pressure, characterizing how busy the demand path is and how close the device is to congestion. CapoTuner maintains Q-values over the discretized cross product of these five signals (Table 1).

Table 1. State signals used by CapoTuner.

State	Measurement	Rationale
`degree`	Current prefetch degree in the window	Current operating point
`accuracy`	$\frac{\#\text{ on-time useful}}{\#\text{ issued}}$	Prefetch quality
`coverage`	$\frac{\#\text{ on-time useful} + \#\text{ inflight}}{\#\text{ demand reads}}$	Prefetch contribution to demand stream
`front_pressure`	$\frac{\text{frontend read BW}}{\text{frontend read BW capacity}}$	Demand-critical-path pressure
`back_pressure`	$\frac{\text{total backend BW}}{\text{backend BW capacity}}$	Backend and CXL path occupancy

Algorithm 1: Q-value-driven Degree Refinement

1. $a^* \leftarrow \textsc{EpsilonGreedy}(Q(s_t, \cdot))$
2. $\Delta Q \leftarrow Q(s_t, a^*) - \max_{a \neq a^*} Q(s_t, a)$
3. $c \leftarrow \textsc{Normalize}(\Delta Q)$
4. **return** $\textsc{Refine}(a^*, c)$

Action. The CapoTuner action space consists of three coarse modes (Low, Moderate, and Aggressive), each mapped to a range of prefetch degrees. The coarse-grained action set lets online learning converge stably within a finite state space and a finite number of windows. As in Athena [8], CapoTuner does not map each mode to a single fixed degree. Instead, CapoTuner uses the normalized Q-value margin between the top action and the runner-up as a confidence value, and uses the confidence to pick a specific degree within the chosen mode's interval. A larger margin selects a more aggressive degree within the interval, and a smaller margin selects a more conservative one. Normalization suppresses the effect of absolute Q-value drift on the confidence estimate.

The coarse action sets the direction, and the Q-value margin chooses the specific degree within the mode's interval (Algorithm 1). CapoTuner internally stores degree state in Q4 encoding (a quarter-degree as the minimum step, with a Q4 value of 4 corresponding to a logical degree of 1), giving the within-interval refinement enough quantization granularity to remain smooth.

Reward. The CapoTuner composite reward separates prefetch benefit from speculative cost explicitly. A single metric cannot reliably capture the true effect of prefetching. Prefetch hit rate alone cannot tell apart prefetches that genuinely help demand accesses from over-prefetches that raise system interference. Absolute latency alone is also unreliable, because the same observed latency can mean different things under different load levels. CapoTuner therefore decomposes the reward at window t into a benefit term R_t^{benefit} and a cost term R_t^{cost}:

$$R_t = R_t^{\text{benefit}} - R_t^{\text{cost}}. \tag{1}$$

The benefit term captures how much prefetching helps the demand path. The cost term captures the interference introduced by the same prefetch traffic. Each term is a weighted sum of device-measurable signals.

The benefit term aggregates a latency-improvement signal and a prefetch-utility signal:

$$R_t^{\text{benefit}} = \lambda_{\text{lat}} M_t^{\text{lat}} + \lambda_{\text{use}} M_t^{\text{use}}. \quad (2)$$

M_t^{lat} is the gain in average demand-read latency observed in the current window relative to the same-load zero-prefetch baseline. M_t^{use} rewards prefetches that genuinely help subsequent demand accesses. The two signals are complementary. M_t^{lat} captures the end-to-end latency benefit visible to the host, and M_t^{use} attributes that benefit to specific prefetches.

The cost term aggregates three contention and waste signals:

$$R_t^{\text{cost}} = \lambda_{\text{load}} M_t^{\text{load}} + \lambda_{\text{write}} M_t^{\text{write}} + \lambda_{\text{waste}} M_t^{\text{waste}}. \quad (3)$$

M_t^{load} penalizes the additional contention introduced by prefetch traffic when the demand-read path is already busy. M_t^{write} separately penalizes prefetch–write contention under high write pressure. M_t^{waste} penalizes prefetches that have been issued but produced no benefit, preventing the controller from trading excessive speculation for short-term gains.

All terms except M_t^{lat} can be derived directly from device-side runtime statistics. M_t^{lat} requires an external reference, which motivates the baseline mechanism described next. Section 5.1 gives the concrete values of the λ coefficients and the tuning procedure.

Latency baseline. CapoTuner maintains a zero-prefetch latency table indexed by demand-read and write bandwidth pressure to compute M_t^{lat}. Each entry records the average demand-read latency at that load point under zero prefetching. At runtime, CapoTuner subtracts the corresponding table entry from the current window's average demand-read latency. The difference becomes M_t^{lat}.

CapoTuner refreshes the baseline table through periodic zero-prefetch probes. Windows in which CapoTuner naturally settles at the zero-prefetch configuration also write directly into the matching entries. A probe stops new prefetch generation while retaining existing data in the PFB. If the current entry has no sample, CapoTuner uses an adjacent write-pressure entry or the row mean. CapoTuner omits the latency term when neither is available. Section 5.1 describes the probe composition and trigger schedule.

Exploration. CapoTuner combines ϵ-greedy with a micro-explore mechanism. ϵ-greedy handles general state–action exploration and prevents the controller from locking into a local optimum too early. Micro-explore targets a degeneration mode specific to CapoTuner. The RL controller can repeatedly receive mediocre negative feedback at zero prefetch and lock itself into the zero-prefetch configuration permanently. After CapoTuner stays at zero prefetch for several consecutive windows, CapoTuner triggers a single probe window at the smallest non-zero prefetch degree.A positive probe outcome restores normal degree selection. Otherwise, the conservative policy continues.

Table 2. Main hardware configuration.

Component	Parameter
CPU	2.4 GHz; 6-wide OoO; 512 ROB entries; 192/128 LQ/SQ entries; L-TAGE; 4096 BTB entries; 32 RAS entries
Core count	1 / 8 / 16 / 32
L1 I/D	Private; 32 KiB 8-way (I) / 48 KiB 12-way (D); 64 B line; LRU; 16 MSHRs (D); 3 cycles
L2 cache	Private; 1 MiB; 64 B line; 16-way; LRU; 48 MSHRs; 48 write buffers; 10 cycles
LLC	2 MiB/core; 64 B line; 16-way; SHiP-PC replacement; 64 MSHRs/slice; 48 write buffers/slice; 96 cycles
Memory	DDR5-4400, single channel each (~35.2 GB/s peak); host DRAM 1 GB, CXL 2 GB
CXL link	PCIe 5.0 ×8 at 32 GT/s; cap enabled; 128/130 encoding; 0.85 protocol overhead; ~26.8 GB/s effective
Prefetch buffer	4 MiB; 16-way; LRU; 20 cycles (Polaris default)
CXL prefetcher	Polaris [44]: virtual-prefetching competitive ensemble over BOP, Domino, VLDP, SPP
CapoMSHR	D-/P-MSHR 48 entries each; queued-candidate table 24 entries

5 EVALUATION

5.1 Experimental Setup

Simulation infrastructure. We evaluate CAPO on SimCXL [41, 42], a full-system cycle-level CXL simulator built on gem5 and calibrated against FPGA/ASIC CXL testbeds with an average simulation error of 3.4% on CXL memory device characteristics. We extend the CXL Type 3 model in two ways. (1) We add a configurable frontend link bandwidth constraint (PCIe lane width selectable). (2) We implement the prefetch unit, prefetch buffer, CapoMSHR, and CapoTuner (Section 4) inside the device controller. All application pages are bound to the CXL memory node via `numactl` (DRAM 1 GB / CXL 2 GB), representing CXL as the main working memory, with the full hardware configuration summarized in Table 2.

Workloads. We evaluate on seven memory-intensive benchmarks: FT (324 MB), CG (187 MB), MG (434 MB), and IS (267 MB) from NAS Parallel Benchmarks (NPB) [3, 6], BFS (557 MB) and PR (558 MB) from the GAP Benchmark Suite [5], and XSBench [38] (501 MB). All benchmarks have LLC MPKI above 6.9 at 1, 8, 16, and 32 cores under the no-prefetch baseline, and all working sets exceed LLC capacity. All workloads run at 1, 8, 16, and 32 cores, with the number of application threads matching the core count.

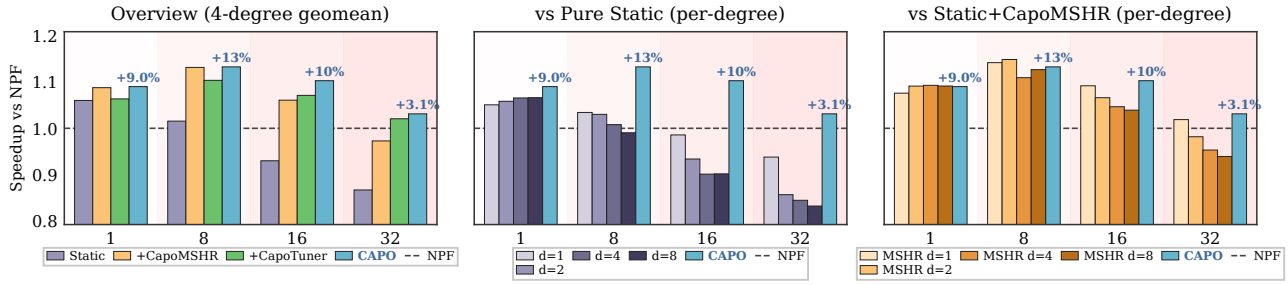


Figure 5. CAPO Outperforms Fixed-Degree Static Prefetching from 1 to 32 Cores.

Table 3. CapoTuner hyperparameters selected by design-space exploration.

Category	Final values
Reward weights	$\lambda_{\text{lat}} = .35$, $\lambda_{\text{use}} = .15$, $\lambda_{\text{load}} = .15$, $\lambda_{\text{write}} = .10$, $\lambda_{\text{waste}} = .04$
RL update	$\alpha = .10$, $\gamma = .90$, $\varepsilon = .10 \rightarrow .01$, decay = .999
Baseline	EMA $\alpha = .20$, probe interval = 32 windows
Window	512 demand reads

Comparison schemes. We evaluate five configurations at 1, 8, 16, and 32 cores. (1) **NPF (CXL-NoPrefetch)** disables device-side prefetching and serves as the primary baseline. (2) **Static** runs Polaris at fixed degree $d \in \{1, 2, 4, 8\}$. (3) **+CapoMSHR** adds CapoMSHR to Static. (4) **+CapoTuner** uses CapoTuner to adjust the Polaris degree online without CapoMSHR. (5) **CAPO** enables both CapoMSHR and CapoTuner. Headline results for Static and +CapoMSHR report the geometric mean over four degrees. +CapoTuner and CAPO adjust degree online and therefore each correspond to one configuration. At 16 and 32 cores, we also compare two device-side rule-based controllers. Near-side throttling (NST) [16] adjusts degree from the late-prefetch fraction, while a link-utilization threshold controller (Threshold) adjusts degree when utilization crosses a fixed threshold. Both use the same Polaris, prefetch buffer, CapoMSHR, and demand-first scheduler as CAPO, with degree caps no lower than CAPO's, so the comparison isolates degree-control policy.

RL hyperparameters. CapoTuner's hyperparameters come from a design-space exploration (DSE) on FT, IS, and CG at 16 cores, covering 3 of the 28 evaluated (workload, core) points, with final values in Table 3. We tune the five reward weights, the window size, ε, and the probe interval by coordinate descent over roughly 100 combinations. We apply the resulting configuration unaltered to all 28 points, and every run starts from an empty Q-table with no pretrained state. Section 5.6 reports held-out and multi-seed results.

Probe execution. CapoTuner inserts two zero-prefetch windows every 32 windows during the same run, one for warm-up and one for measurement. Both windows count toward reported IPC. Neither updates Q-values, and only the measurement window updates the baseline table through an exponential moving average.

Sampling. Each workload executes 10–50 billion instructions after initialization, beyond the feasible range of full-system O3 simulation, so we use LoopPoint [29] sampling. At 8 cores, the basic block vector (BBV) region length is 80 M committed instructions, scales with core count, and excludes synchronization-library instructions. We use SimPoint [31] for k-means clustering, with k selected by BIC over $[3, 30]$ and final values in $[4, 12]$. Each region is preceded by an equal-length warm-up region. Each region runs as an independent gem5 process, so learned tables do not persist across regions. CapoTuner learns throughout the warm-up region. Measurement resets only performance statistics while the learned state persists and continues updating. Median measurement length ranges from 4.7K to 12.3K windows across 8–32 cores. Per-region results are aggregated by cluster weight. On several short workloads, LoopPoint sampling has less than 5% relative IPC error against full-length detailed simulation.

5.2 Main Performance

We report speedup as the geometric mean over the seven workloads relative to NPF, computed from per-benchmark IPC (committed instructions over simulated cycles, averaged arithmetically across cores).

Overall speedup. CAPO achieves a geometric-mean speedup over NPF of +9.0%, +13.3%, +10.3%, and +3.1% at 1, 8, 16, and 32 cores (Figure 5, Overview panel). Under the same configuration the four-degree geometric mean of Static deteriorates to −7.0% at 16 cores and −13.3% at 32 cores, and CAPO converts these losses into positive speedups.

Contribution of each component. CapoMSHR and CapoTuner play different roles across core counts. +CapoMSHR alone delivers +8.8%, +13.2%, and +6.1% at 1, 8, and 16 cores, almost matching CAPO at 1 and 8 cores. At 32 cores +CapoMSHR drops to −2.7% and cannot sustain high concurrency on its own. +CapoTuner alone yields +6.4%, +10.4%, +7.1%, and +2.1% across the four core counts,

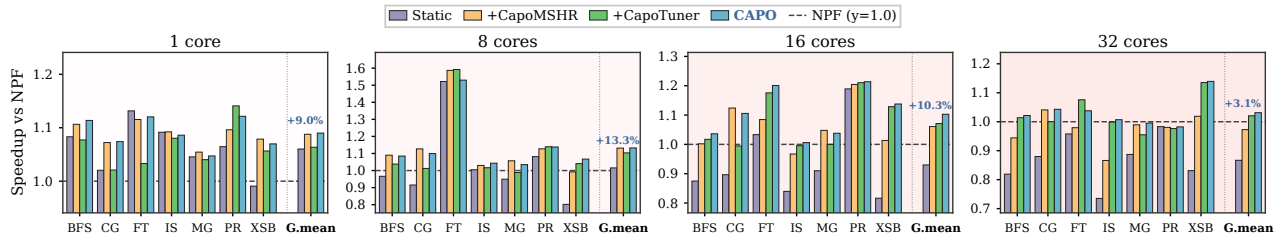
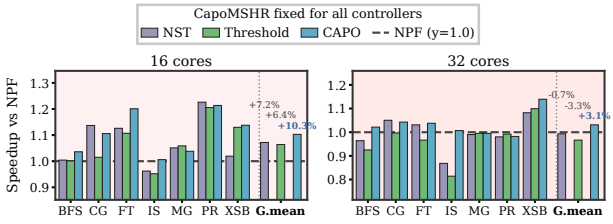
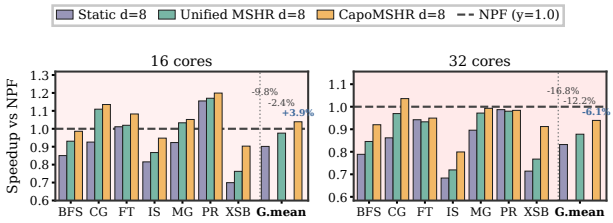


Figure 6. CAPO Improves Most Workload-Core Points.

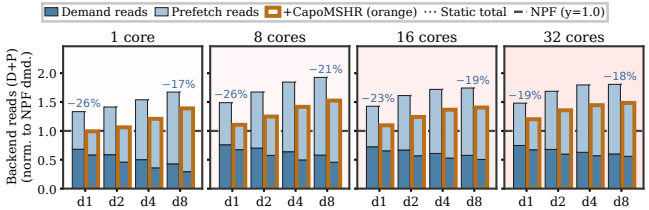


(a) NST and Threshold turn negative at 32 cores.

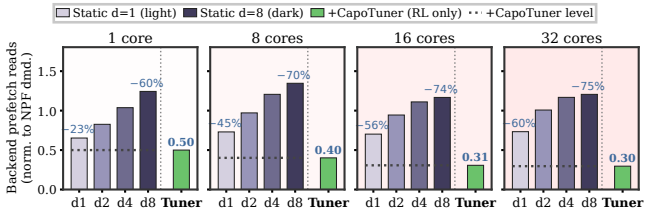


(b) A unified MSHR falls to −12.2% at 32 cores.

Figure 7. CAPO Outperforms Rule-Based and Unified Alternatives.



(a) CapoMSHR cuts backend reads by 17–26%.



(b) CapoTuner cuts backend prefetch reads by 23–75%.

Figure 8. CapoMSHR and CapoTuner Reduce Backend Traffic.

staying positive everywhere but trailing +CapoMSHR at 1 and 8 cores. CAPO combines the two and is positive across the full range, including the +3.1% at 32 cores that +CapoMSHR alone cannot reach. Section 5.5 quantifies the interplay between the two components.

Comparison against simpler controllers. CAPO stays above the best fixed-degree Static at all four core counts (Figure 5, middle panel) and leads Static's four-degree geometric mean by 17.3 and 16.4 percentage points at 16 and 32 cores. No fixed degree covers the range: the best degree for +CapoMSHR shifts from d=4–8 at 1 core to d=1 at 32 cores (Figure 5, right panel). Single-signal rules also fall short. At 16 cores, NST and Threshold reach 7.2% and 6.4% against CAPO's 10.3%, and CAPO is the only controller positive on all seven workloads (Figure 7a). At 32 cores, both rules turn negative, at −0.7% and −3.3%, while CAPO retains +3.1%. NST and Threshold use the late-prefetch fraction and link utilization, respectively, while CAPO adjusts degree jointly from prefetch effectiveness and frontend and backend pressure.

Distribution across workloads. CAPO is positive on 26 of the 28 (workload, core) points (Figure 6), with the maximum single-point gain of +52.98% on FT at 8 cores. The two negative points are PR (−1.78%) and MG (−0.48%), both at 32 cores, with attribution discussed in Section 5.6. The 26 positive points span all four core counts and all seven workloads, so the geometric-mean gain does not rest on a few large outliers.

5.3 CapoMSHR Mechanism

Backend-read savings. CapoMSHR reduces total backend reads by 17–26% at every (core, d) point, and by 18–23% at 16 and 32 cores (Figure 8a). Both segments contribute: reusing an inflight prefetch response removes a redundant demand read, and discarding a candidate whose address is already tracked removes a prefetch before it issues. The prefetch segment carries 52–87% of the saving. CapoMSHR never changes the degree, so the absolute reduction stays at 0.28–0.43× the NPF demand volume across the full degree range.

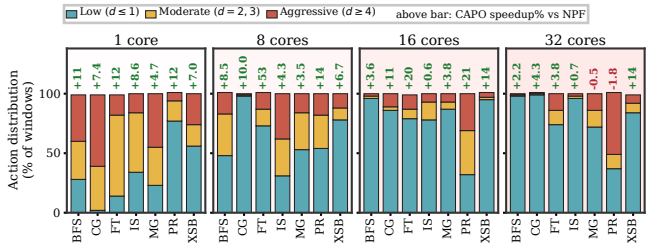


Figure 9. CapoTuner Lowers Prefetch Intensity as Core Count Increases.

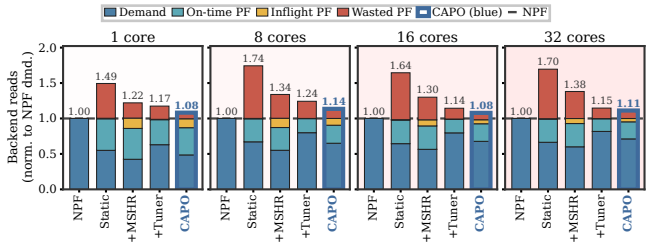


Figure 10. CAPO Keeps Backend Traffic Low and Raises Useful Prefetches.

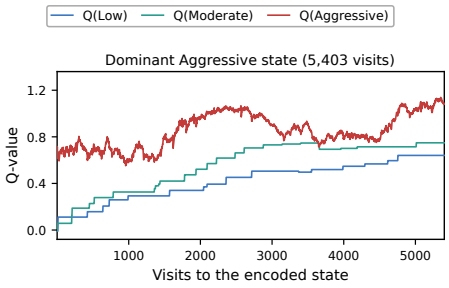


Figure 11. PR's Dominant State Keeps Preferring Aggressive at 32 Cores.

Split versus unified tracking. Demand–prefetch coordination can also use a unified MSHR, so we isolate the effect of the split organization. We compare CapoMSHR against a conventional 288-entry unified MSHR at Static d=8, more than twice the 120 entries of CapoMSHR's split structures combined. CapoMSHR outperforms the unified design by 6.5% and 7.0% at 16 and 32 cores (Figure 7b). Relative to NPF, the unified design falls from −2.4% at 16 cores to −12.2% at 32 cores as demand and prefetch entries contend for one table. Split tracking prevents prefetch entries from crowding out demand entries while retaining demand–prefetch coordination.

Backend-read savings alone still do not make Static positive at 32 cores, where overly aggressive prefetch decisions remain (Section 5.4).

5.4 CapoTuner Mechanism

Prefetch-traffic suppression. +CapoTuner reduces backend prefetch-read traffic below the corresponding Static at every (core, d) comparison point, with the reduction increasing from 23% at 1 core and d=1 to 75% at 32 cores and d=8 (Figure 8b).

Prefetch-degree selection. We group CapoTuner's per-window actions into Low ($d \in [0, 1]$), Moderate ($d \in [2, 3]$), and Aggressive ($d \in [4, 8]$). At 1 core, BFS, CG, and MG select Aggressive in more than 30% of windows. By 32 cores, almost all workloads are dominated by Low and Moderate (Figure 9). The shift toward lower intensity directly reduces issued prefetch requests, matching the traffic reduction under high concurrency.

5.5 Combined Contribution

Each backend read is classified as demand, on-time prefetch, inflight prefetch, or wasted prefetch, normalized to the NPF demand-read count and stacked (Figure 10). Static and +CapoMSHR are aggregated as the arithmetic mean over $d \in \{1, 2, 4, 8\}$.

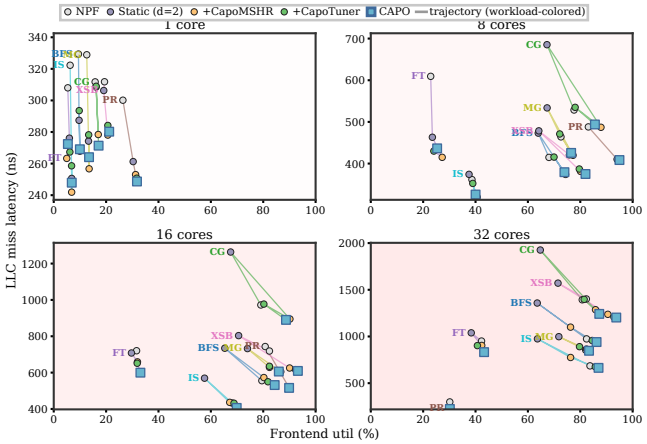
Total traffic convergence. The Static stack reaches 1.49–1.74× NPF and stays above 1.6× beyond 8 cores. +CapoMSHR brings it down to 1.22–1.38×, with most of the reduction in the prefetch segment. +CapoTuner alone brings it down to 1.14–1.24×, with the savings concentrated in the wasted-prefetch segment. CAPO combines the two and reaches 1.08–1.14× NPF, trading 8–14% extra backend traffic for the +3.1%–+13.3% geometric-mean speedup of Section 5.2.

Useful-fraction reversal. The useful fraction (on-time plus inflight) of Static is only 31–48%. +CapoMSHR raises it to 51–73% by absorbing overlapping demands and dropping duplicate candidates, and +CapoTuner reaches 45–66% by issuing fewer prefetches. CAPO combines both routes and stabilizes it at 72–85%, issuing fewer but higher-hit-rate prefetches.

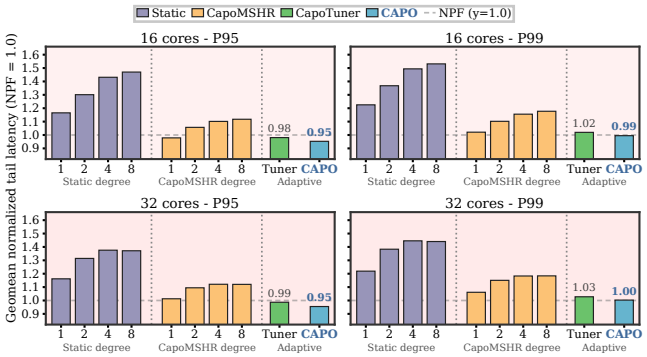
Non-orthogonality. CapoMSHR's effectiveness depends on an inflight prefetch pool to which demands can attach. CapoTuner directly shrinks that pool, so demands that could have been absorbed by merging instead pay the full CXL round-trip latency. At 32 cores, adding CapoMSHR on top of +CapoTuner therefore contributes only about one percentage point. The two mechanisms cover different pressure regimes, so their gains overlap rather than add.

5.6 Generalization and Failure Attribution

Held-out workloads. The DSE uses only 3 of the 28 (workload, core) points (FT, IS, and CG at 16 cores). Sixteen of the remaining 25 points come from four workloads (BFS, MG, PR, and XSBench) that the DSE never sees at any core count, a workload-level holdout. On those 16 points CapoTuner alone gains +5.8% and CAPO +7.6%, with CAPO positive on 14 of the 16.



(a) CAPO sits lower and further right than Static.



(b) CAPO holds P95 and P99 at NPF parity.

Figure 12. CAPO Reduces Latency Without Tail Inflation.

Failure attribution. Both of CAPO's negative points occur at 32 cores. PR at 32 cores still selects Aggressive in roughly half of its windows. Among those selections, 99.4% follow the highest Q-value rather than exploration, and Q(Aggressive) stays highest across all 5,403 visits to the dominant state (Figure 11). The coarse state encoding maps scattered multi-core access onto the same Q-entry as other high-concurrency regimes, so the learned Aggressive preference carries into windows where it does not pay. Across three seeds, PR at 32 cores has a mean of −0.5% with a standard deviation of 2.5 percentage points, so its sign is unstable. MG has a mean of −1.0% with a standard deviation of 0.5 points. Section 5.2 reports one seed from each set.

5.7 Latency Analysis

Latency versus throughput. CAPO's markers sit to the lower right of NPF, Static, +CapoMSHR, and +CapoTuner on almost every workload trajectory (Figure 12a), with frontend utilization (demand-read bytes on the CXL upstream link over link capacity) on the x-axis and LLC miss end-to-end latency on the y-axis. Device-side prefetching leaves the host LLC untouched, so every configuration moves the same demand-read volume for a given workload and core count, so higher utilization means that volume completes in less time. A pending LLC miss stalls dependent instructions, so lower latency lets the CPU issue its next requests sooner. CAPO's position therefore reflects both lower latency and the throughput it releases, while Static sits at the upper left, in line with the speedups of Section 5.2 and the traffic decomposition of Section 5.5.

Table 4. System sensitivity on FT and BFS at 32 cores. *marks the default configuration.

Link / backend	CAPO	Static d=4	CAPO vs. Static
×8 / DDR5-4400*	+2.4%	−14.1%	+19.2%
×4 / DDR5-4400	−1.4%	−6.4%	+5.3%
×8 / DDR5-3200	+3.5%	−20.1%	+29.5%
×4 / DDR5-3200	+2.2%	−7.3%	+10.3%

PFB capacity	1 MiB	2 MiB	4 MiB*	8 MiB	16 MiB
CAPO vs. NPF	−1.5%	−1.6%	+2.4%	+1.4%	+2.1%

Tail latency. Window-averaged demand-read P95 and P99 at the CXL controller tell the same story (Figure 12b). At 16 cores Static d=1–8 inflates them to 1.17–1.47× and 1.23–1.53× NPF, and at 32 cores to 1.16–1.38× and 1.22–1.45×. CAPO holds P95 at 0.95× and P99 at 0.99–1.00× at both core counts, removing Static's tail inflation while keeping P99 parity with NPF.

5.8 System Sensitivity

The main configuration uses PCIe 5.0 ×8, DDR5-4400, and Polaris's default 4 MiB PFB, and every IPC change below is measured against the NPF baseline of its own configuration. Under reduced backend bandwidth (DDR5-3200), Static d=4 loses 20.1%, while CAPO gains 3.5% (Table 4). When link bandwidth is halved, the congestion mechanism weakens, yet CAPO still leads Static by 5.3%. CAPO stays positive from 4 to 16 MiB PFB without a monotonic capacity benefit. A 1 or 2 MiB PFB erases CAPO's absolute gain in these workloads.

Probe interval. Sweeping the probe interval over 8, 32, and 64 windows on FT and BFS at 32 cores moves the speedup by at most 0.6 percentage points, so degree selection is insensitive to the probe cadence.

5.9 Overhead Analysis

The on-device control state totals 19.76 KiB: 1.10 KiB CapoMSHR, 3.38 KiB CapoTuner, and 15.28 KiB for the Polaris prefetch unit, which is also present in the static-prefetch baseline. The two CAPO structures account for 4.48 KiB. The two figures are 0.48% and 0.11% of the 4 MiB prefetch buffer. The CapoMSHR per-entry width stays below classical LLC MSHRs because the split design needs no

demand–demand merging, and the reward computation runs at window boundaries, off the demand critical path.

6 DISCUSSION

Two system-level interactions arise when adaptive device-side prefetching moves into broader data-center deployments, and we plan to address both in future work.

Multi-tenant and pooled devices. CAPO currently targets a single-host CXL Type 3 expander, where the controller observes a unified demand stream and aggregates device-wide reward metrics. In pooled memory systems connected via CXL switches, multiple hosts share the backend memory. Such device-wide aggregation blurs cross-tenant contention boundaries: aggressive prefetches issued for one tenant consume bandwidth that demand requests from other tenants need. CapoMSHR carries the same limitation on the tracking side, since its P-MSHR records inflight lines without host tags, leaving per-tenant isolation and fairness out of reach. Prefetch control under pooling therefore needs tenant-aware MSHR tracking and a QoS-guided reward.

Interaction with page migration. Our evaluation binds the entire working set to CXL memory. In tiered memory hierarchies, page migration moves hot pages to local DRAM, affecting device-side prefetching in two ways. (1) Migrating high-reuse pages filters the demand stream, leaving a residual pattern with sparser reuse and weaker spatial correlation. (2) Bulk page copies inject bursty, one-off sequential reads that distort online pattern learning. These transfers are indistinguishable from regular demand accesses at the device side, so coordinating with host-driven migration requires exposing migration as a lightweight event on the CXL link.

7 RELATED WORK

7.1 Device-side prefetching for memory expanders

Three prior works study prefetching inside the memory or storage device controller. Polaris [44] introduces an ensemble prefetch unit at the CXL memory controller, with a virtual competition over BOP, Domino, VLDP, and SPP at a fixed prefetch degree. CAPO uses the same Polaris ensemble as its prefetch backbone (Section 4). LG-Prefetcher [22] adjusts the prefetch range from accuracy, coverage, lateness, and pollution feedback in the CXL-SSD setting. Yang et al. [43] explore device-side prefetching as part of a broader CXL-SSD design study, tuning aggressiveness from accuracy and a dynamic window. None of the three exposes link bandwidth pressure as a control signal, and none provides demand–prefetch inflight coordination inside the device.

7.2 Prefetch throttling and adaptive control

Prefetch throttling research on the CPU side splits into two design families. Heuristic feedback controllers adjust aggressiveness from prefetcher-quality signals. FDP [33] feeds back accuracy, lateness, and pollution, CMPP [11] coordinates multiple prefetchers, and near-side throttling [16] drives prefetch distance from the late-prefetch fraction. Online-learning controllers instead learn a policy. Pythia [7] runs online RL over program features and host-side bandwidth signals, Athena [8] pairs coordination between the prefetcher and an off-chip predictor (OCP) with expected-reward-driven aggressiveness adjustment, and a multi-armed bandit prefetcher [13] uses discounted UCB over a 100 B state. All of these controllers run inside the CPU and rely on host microarchitectural state such as L2 cache evictions, per-core MSHR occupancy, and memory bandwidth utilization. CapoTuner ports the online-learning paradigm to the CXL device side, where the controller observes only aggregated multi-core traffic without any host visibility and trades link bandwidth pressure against the latency-hiding benefit of prefetching.

To our knowledge, CAPO is the first device-side CXL prefetch controller that combines inflight demand–prefetch coordination with online load-aware degree adaptation, occupying a design point that prior device-side proposals (no adaptive control) and prior adaptive prefetchers (no device-side deployment) do not cover.

8 CONCLUSION

We present CAPO (CXL Adaptive Prefetch cOntroller), an adaptive prefetch controller for CXL Type-3 memory expansion devices. CAPO comprises two components. CapoMSHR tracks demand and prefetch requests separately inside the CXL controller and eliminates redundant backend reads when a demand hits an inflight prefetch. CapoTuner uses online reinforcement learning to set the prefetch degree each window based on device-observable signals of load, latency, and prefetch utility, and tracks the right operating point across workloads and concurrency levels. Across seven memory-intensive workloads and 1 to 32 cores on SimCXL, CAPO achieves a 3.1%–13.3% geometric-mean speedup over the no-prefetch baseline. At 16 and 32 cores, CAPO turns the −7.0% and −13.3% geometric-mean slowdowns of fixed-degree static prefetching into +10.3% and +3.1% speedups. Beyond the Polaris-style ensemble evaluated here, CAPO's two-component pattern of device-side inflight tracking and load-aware online control provides a reusable substrate for adaptive prefetching in memory-expansion devices without host visibility.

References

[1] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 775–787. https://www.usenix.org/conference/atc18/presentation/aguilera

[2] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching.

In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 513–526. doi:10.1145/3373376.3378498

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73. doi:10.1177/109434209100500306

[4] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino Temporal Data Prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 131–142. doi:10.1109/HPCA.2018.00021

[5] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. *arXiv preprint arXiv:1508.03619* (2015). arXiv:1508.03619 [cs.DC] https://arxiv.org/abs/1508.03619

[6] Benchmark Subsetting. 2014. NPB3.0-omp-C: NAS Parallel Benchmarks 3.0 OpenMP C Version. https://github.com/benchmark-subsetting/NPB3.0-omp-C. Unofficial OpenMP C version of NAS Parallel Benchmarks 3.0.

[7] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1121–1137. doi:10.1145/3466752.3480114

[8] Rahul Bera, Zhenrong Lang, Caroline Hengartner, Konstantinos Kanellopoulos, Rakesh Kumar, Mohammad Sadrosadati, and Onur Mutlu. 2026. Athena: Synergizing Data Prefetching and Off-Chip Prediction via Online Reinforcement Learning. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1–19. doi:10.1109/HPCA68181.2026.11408449

[9] CXL Consortium. 2025. *Compute Express Link (CXL) 4.0 Specification*. https://computeexpresslink.org/cxl-specification/ Accessed: 2026-04-21.

[10] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* 56, 11, Article 290 (July 2024), 37 pages. doi:10.1145/3669900

[11] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 316–326. doi:10.1145/1669112.1669154

[12] K. I. Farkas and N. P. Jouppi. 1994. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (Chicago, Illinois, USA) *(ISCA '94)*. IEEE Computer Society Press, Washington, DC, USA, 211–222. doi:10.1109/ISCA.1994.288148

[13] Gerasimos Gerogiannis and Josep Torrellas. 2023. Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making. In *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 698–713.

[14] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu

[15] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who limits the resource efficiency of my datacenter: an analysis of Alibaba datacenter traces. In *Proceedings of the International Symposium on Quality of Service* (Phoenix, Arizona) *(IWQoS '19)*. Association for Computing Machinery, New York, NY, USA, Article 39, 10 pages. doi:10.1145/3326285.3329074

[16] Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. 2018. Near-side prefetch throttling: adaptive prefetching for high-performance many-core processors. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 28, 11 pages. doi:10.1145/3243176.3243181

[17] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *2008 International Symposium on Computer Architecture*. 39–50. doi:10.1109/ISCA.2008.21

[18] Bruce Jacob. 2009. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Springer.

[19] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A.L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. doi:10.1109/MICRO.2016.7783763

[20] David Kroft. 1981. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture* (Minneapolis, Minnesota, USA) *(ISCA '81)*. IEEE Computer Society Press, Washington, DC, USA, 81–87. https://dl.acm.org/doi/10.5555/800052.801868

[21] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. doi:10.1145/3575693.3578835

[22] Shibao Li, Chengzhi Wang, Zhou Yang, Yunwu Zhang, Longfei Li, Chenxu Ma, Xuerong Cui, Lianghai Li, and Jianhang Liu. 2025. LG-Prefetcher: A Prefetcher Combining Local Spatial and Global Performance Information for CXL-SSD. *The Journal of Supercomputing* 81, 1430 (2025). doi:10.1007/s11227-025-07910-7

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

[24] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 742–755. doi:10.1145/3582016.3582063

[25] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 469–480. doi:10.1109/HPCA.2016.7446087

[26] Sparsh Mittal. 2016. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Comput. Surv.* 49, 2, Article 35 (Aug. 2016), 35 pages. doi:10.1145/2907071

[27] Vinicius Petrucci, Eishan Mirakhur, Rita Gupta, Mahesh Wagh, Nikesh Agarwal, Su Wei Lim, and Vishal Tanna. 2025. CXL Memory Expansion: A Closer Look on Actual Platform. https://www.micron.com/content/dam/micron/global/public/products/white-paper/cxl-memory-expansiona-close-look-on-actual-platform.pdf. Micron White Paper.

[28] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[29] Alen Sabu, Harish Patil, Wim Heirman, and Trevor E. Carlson. 2022. LoopPoint: Checkpoint-driven Sampled Simulation for Multi-threaded Applications. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 604–618. doi:10.1109/HPCA53966.2022.00051

[30] Debendra Das Sharma. 2023. Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy. *IEEE Micro* 43, 2 (2023), 99–109. doi:10.1109/MM.2022.3228561

[31] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 45–57. doi:10.1145/605397.605403

[32] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 141–152. doi:10.1145/2830772.2830793

[33] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 63–74. doi:10.1109/HPCA.2007.346185

[34] Yan Sun, Jongyul Kim, Zeduo Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2025. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 604–621. doi:10.1145/3676641.3711999

[35] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 105–121. doi:10.1145/3613424.3614256

[36] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2 ed.). The MIT Press, Cambridge, MA.

[37] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. doi:10.1145/3342195.3387517

[38] John R. Tramm, Andrew R. Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench – The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *Proceedings of PHYSOR 2014: The Role of Reactor Physics toward a Sustainable Future*. Kyoto, Japan. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[39] James Tuck, Luis Ceze, and Josep Torrellas. 2006. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 409–422. doi:10.1109/MICRO.2006.44

[40] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. 2024. Designing Cloud Servers for Lower Carbon. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 452–470. doi:10.1109/ISCA59077.2024.00041

[41] Yanjing Wang, Lizhou Wu, Sunfeng Gao, Yibo Tang, Junhui Luo, Zicong Wang, Yang Ou, Dezun Dong, Nong Xiao, and Mingche Lai. 2026. Cohet: A CXL-Driven Coherent Heterogeneous Computing Framework with Hardware-Calibrated Full-System Simulation. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 1–16. doi:10.1109/HPCA68181.2026.11408611

[42] Yanjing Wang, Lizhou Wu, Wentao Hong, Yang Ou, Zicong Wang, Sunfeng Gao, Jie Zhang, Sheng Ma, Dezun Dong, Xingyun Qi, Mingche Lai, and Nong Xiao. 2025. CXL-DMSim: A Full-System CXL Disaggregated Memory Simulator With Comprehensive Silicon Validation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2025), 1–1. doi:10.1109/TCAD.2025.3607145

[43] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. 2023. Overcoming the Memory Wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 601–617. https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

[44] Zhe Zhou, Shuotao Xu, Yiqi Chen, Tao Zhang, Ran Shu, Lei Qu, Peng Cheng, Yongqiang Xiong, and Guangyu Sun. 2024. Polaris: Enhancing CXL-based Memory Expanders with Memory-side Prefetching. In *Advanced Parallel Processing Technologies*, Chao Li, Zhenhua Li, Li Shen, Fan Wu, and Xiaoli Gong (Eds.). Springer Nature Singapore, Singapore, 19–39. doi:10.1007/978-981-99-7872-4_2