PACT: A Criticality-First Design for Tiered Memory

Hamid Hadian
Virginia Tech
Blacksburg, USA

Jinshu Liu*
Virginia Tech
Blacksburg, USA

Hanchen Xu*
Virginia Tech
Blacksburg, USA

Hansen Idden
Virginia Tech
Blacksburg, USA

Huaicheng Li
Virginia Tech
Blacksburg, USA

Abstract

Tiered memory systems typically place pages based on access frequency (hotness), yet frequency alone fails to capture the true performance impact. We present PACT, an online, page-granular tiered memory design that elevates performance criticality to a first-class design principle. At its core is Per-page Access Criticality (PAC), a fine-grained metric that quantifies each page's contribution to application performance rather than merely counting accesses. PACT profiles PAC online using a lightweight analytical model that uniquely decomposes per-tier *memory-level parallelism via hardware queue occupancy counters, enabling direct CPU stall attribution to individual pages. To handle highly skewed PAC distributions, PACT employs PAC-centric migration policies: eager demotion and adaptive promotion, to dynamically place performance-critical pages in DRAM. Across 13 workloads, PACT achieves up to 61% performance improvement over the best of 7 state-of-the-art tiering designs with up to 50× fewer migrations.*

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

Keywords: Tiered Memory, Compute Express Link (CXL), Operating Systems

ACM Reference Format:
Hamid Hadian, Jinshu Liu, Hanchen Xu, Hansen Idden, and Huaicheng Li. 2026. 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, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3779212.3790198

1 Introduction

The growing gap between compute performance and memory capacity has made tiered memory architectures essential for modern datacenters. As DRAM scaling slows and memory-hungry workloads continue to grow, systems increasingly combine fast-tier memory (DRAM) with slow-tier alternatives (NUMA, persistent memory, and CXL) [23, 30, 37, 45]. Compute Express Link (CXL) accelerates this trend by enabling hardware-level memory disaggregation and pooling, but CXL access latencies remain 2–3× higher [3, 19, 21, 32, 47, 48]. This latency gap makes effective tiered memory management critical for performance-sensitive applications.

Existing tiered memory systems address this challenge through page sampling, allocation, and migration techniques that promote "hot" pages to fast-tier memory [17, 24, 25, 27, 29, 37, 44, 51–53, 56, 58, 59]. These systems rely on *hotness*, typically page-level access frequency, to guide placement decisions, assuming frequently accessed pages are performance-critical.

However, hotness is an unreliable proxy for performance impact [8, 22, 34]. Memory access criticality, defined as the performance cost an access imposes on the CPU, depends on many factors, such as access patterns, memory-level parallelism (MLP), and access latency, rather than access frequency alone [20, 22, 34, 51]. For instance, sequential accesses with high MLP (*e.g.*, array traversals) can tolerate slow-tier latency with minimal performance impact, while pointer-chasing operations with low MLP suffer proportional slowdowns [22, 34]. This fundamental disconnect, that frequency does not equal criticality, motivates our work.

While recent work has moved beyond access frequency toward criticality, these approaches either rely on offline, coarse-grained profiling (*e.g.*, object level) or incorporate criticality only as a reactive hint layered atop fundamentally hotness-driven policies [22, 34, 52]. Consequently, criticality is never elevated to a first-class design principle, leaving these systems without online adaptability or page-level precision necessary for effective tiered memory management.

Our key insight is that effective page placement demands a criticality-first redesign in which the runtime performance impact of each page access is directly quantified, rather than inferred through indirect proxies. To this end, we introduce **Per-page Access Criticality** (**PAC**), a metric that quantifies each page's contribution to CPU stall time online through an accurate, *per-tier* MLP decomposition, derived from a study of 96 workloads. PAC provides fine-grained precision at both

*Equal contribution.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790198

4KB pages and hugepages, while adapting online to evolving memory access patterns. By closing the semantic gap between measurement and migration granularity, PAC enables a fundamentally new class of page placement and migration policies grounded in true performance cost. We present **PACT**, the first tiered memory system designed around PAC, in which online, page-granular performance criticality directly drives page placement and migration decisions.

Realizing this criticality-first design requires overcoming two core challenges: *(1) Accurate PAC estimation*: How can we quantify the performance impact of each page access online without direct hardware support? *(2) PAC-centric policies*: How can we leverage PAC to design efficient and robust sampling, promotion, and demotion mechanisms that remain effective across diverse workloads? While CPUs expose stall-related performance counters that correlate with application performance, these counters are coarse-grained, per-core metrics that aggregate memory access effects across memory tiers and lack the per-page, per-tier granularity required for PAC [19, 30, 32–36, 50].

Nevertheless, we demonstrate that accurate PAC estimation is achievable through careful analytical modeling using only four standard CPU performance counters. Our large-scale study of 96 real-world workloads across three latency configurations reveals two key insights (more in §4.2). First, *per-tier CPU stalls can be accurately modeled* as a function of LLC-misses, latency, and MLP, achieving a Pearson correlation coefficient above 0.98. Second, *workloads exhibit stable execution phases*, in which MLP remains consistent for tens of milliseconds, while evolving over time to reflect changes in memory access patterns. This phase stability enables accurate per-page attribution of per-tier stall time proportional to access frequency within a small time window, while still capturing dynamic shifts in performance criticality over time. Together, these insights enable PACT to estimate and adapt PAC online, providing a practical foundation for criticality-driven tiered memory management.

Building on this foundation, PACT is a lightweight tiering design that migrates pages based on their true performance impact. Operating transparently without requiring application changes, PACT periodically samples memory accesses and accumulates PAC distributions to drive background migrations. Its runtime engine efficiently performs sampling, ranking, and PAC maintenance. To address highly skewed PAC distributions (§3) and balance migration overhead against performance gains, PACT employs two PAC-centric strategies: *eager demotion* and *adaptive promotion*. Notably, eager demotion proactively frees space in the fast tier to ensure timely promotion of high-PAC pages. Adaptive promotion leverages lightweight statistical sampling with adaptive binning to efficiently identify high-PAC candidates, eliminating the need for global sorting or static thresholds, and thereby substantially reducing manual tuning.

We evaluate PACT on a diverse suite of 13 memory-intensive workloads spanning graph analytics, HPC, in-memory caching, and machine learning, under a wide range of system configurations. We compare against 7 prior tiering systems across 7 fast/slow tier ratios: Soar and Alto [34], Memtis [29], Colloid [51], Nomad [53], TPP [37], and Linux NUMA Balancing Tiering (NBT) [2]. Across these settings, PACT achieves strong performance improvements, by up to 61% over the second-best system. In cases where PACT is not the top performer, it closely tracks the best-performing baseline, with an average gap of 4.1% and a maximum gap of 11.8%. In addition, PACT reduces migrations while requiring minimal parameter tuning. Extensive sensitivity analyses further demonstrate that PACT remains robust across workload characteristics and system configurations.

In summary, this paper makes the following contributions:

- We establish *page-granular, online performance criticality* as a first-class design principle for tiered memory management, moving beyond frequency-based and coarse-grained cost-aware approaches.
- We introduce PAC, a metric that quantifies each page's contribution to CPU stalls via MLP-aware, per-tier performance modeling using only 4 CPU performance counters.
- We design PACT, the first *online, page-granular, criticality-first* tiered memory system, with novel PAC-centric policies for eager demotion and adaptive promotion tailored to highly skewed PAC distributions.
- We evaluate PACT against 7 state-of-the-art tiering designs and demonstrate clear performance advantages and robustness across workloads and configurations, establishing online performance criticality as a practical foundation for efficient tiered memory management.
- We open-source PACT at https://github.com/MoatLab/PACT.

The rest of this paper is organized as follows: §2 provides background and related work. §3 motivates PAC through empirical analysis. §4 presents PACT design. §5 evaluates PACT against state-of-the-art, and §6 concludes.

2 Background and Related Work

In this section, we contrast PAC and PACT with prior hotness-based and cost-aware designs to clarify the unique capabilities of PAC and the new policy design space it enables.

2.1 Hotness vs. Performance Criticality

Hotness-based designs implicitly assume that frequently or recently accessed pages are performance-critical for fast-tier placement [28, 29, 37, 44, 46, 51, 53, 56, 58, 59]. However, not all memory accesses have equal performance impact, as modern CPUs effectively hide latency via out-of-order executions [34]. For example, today's processors exploit MLP to issue concurrent memory requests to amortize the memory latency impact [20]. Consider two pages with identical access frequency: one accessed during pointer-chasing (serialized) and another during array streaming (concurrent). The pointer-chasing page exposes the full memory latency per

access, causing significant CPU stalls, while the streaming page amortizes latency across overlapping requests, reducing per-access stall cost. Despite equal frequency, these pages have vastly different performance criticality.

2.2 PACT vs. State-of-the-Art

Moving beyond hotness, SoarAlto [34] is the most closely related work in demonstrating that Amortized Offcore Latency (AOL) predicts performance more accurately than frequency. However, it operates at coarse object granularity and relies on offline profiling. In practice, pages within the same object can exhibit sharply different and dynamically evolving criticality profiles (§3). This granularity mismatch between profiling granularity and migration granularity, together with the lack of online adaptation, fundamentally limits existing criticality-aware approaches, motivating the need for fine-grained, online performance tracking.

In more detail, while both AOL and PAC account for MLP when estimating memory access cost, they differ fundamentally in mechanism and scope. AOL computes amortized latency as $\frac{\text{Latency}}{\text{MLP}}$ using *system-wide* MLP from offline profiling, yielding per-object or per-period criticality estimates that cannot distinguish between memory tiers at runtime. In contrast, PAC decomposes MLP *per-tier* by leveraging occupancy information from the CPU's CHA/TOR queues located between the cores and DRAM/CXL, enabling online estimation of each page access's contribution to slow-tier stalls (§4.2). This per-tier decomposition is essential for tiering memory systems, as it allows PACT to quantify the benefit of promoting a specific page from the slow tier to the fast tier, rather than relying on aggregate object-level estimates that obscure page-level and tier-level variations.

Similarly, while CPU stalls are a natural indicator of performance impact and prior works such as TMO have leveraged stall-related pressure signals [52], these signals are used as reactive system-level indicators rather than as foundational design abstractions. In contrast, PACT elevates fine-grained online performance criticality to the core of its policy engine. By enabling precise identification of which specific pages bottleneck performance, PAC allows PACT to make targeted placement decisions that are not possible with prior stall-, pressure-, or hotness-based systems [22, 31, 34, 51, 52].

2.3 Tiering Mechanisms and Policies

Hotness tracking. Conventional methods such as page table scanning or poisoning provide fine-grained visibility but incur high overhead from TLB shootdowns and fault handling [23, 37, 41, 53]. Hardware-assisted sampling (*e.g.*, Intel PEBS) offers low-cost monitoring via LLC miss events [1, 6, 29, 44], but lacks semantic context of accesses. More recently, both academia and industry have explored controller-side hotness tracking, though such support is not yet commercially available [3, 43, 46, 59]. While these approaches are effective for tracking frequency, they do not capture the true performance cost of each access. In contrast, PACT pursues a new direction: tracking per-page performance criticality, *i.e.*, how much each access stalls the CPU, which demands a fundamentally different sampling design.

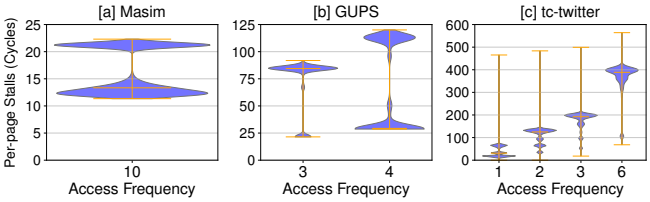

Figure 1. PAC vs. frequency. *Each violin plot shows the distribution of PAC in cycles across page access frequency quantiles (x-axis).*

Tiering policies. Existing tiering policies struggle with deciding which pages to migrate, when, and how aggressively. First, hot pages are promoted regardless of whether those accesses impact performance. Second, frequency-based heuristics trigger excessive migrations whose costs outweigh the benefits. Third, most policies lack adaptivity, aggressively migrating pages even under pressure. These limitations lead to unnecessary migrations, poor fast-tier utilization, and excessive overhead, often negating the benefits of tiering (§5). PACT addresses these challenges with a PAC-centric design that drives flexible, performance-aware tiering. By coupling eager demotion with adaptive promotion, PACT adapts to workload dynamics while minimizing migration overhead.

3 Motivation

To quantitatively demonstrate how criticality diverges from frequency, we profile three applications on an emulated CXL device with 190ns latency (§5): Masim [9], GUPS [4], and tc-twitter from GAPBS [13]. These benchmarks span diverse memory access patterns and computational characteristics. Our measurements reveal a clear disconnect between frequency and performance impact. Pages with identical access frequencies can differ in criticality by as much as 65×, demonstrating that frequency is insufficient to capture true cost and motivating a shift toward criticality-based placement.

Methodology. We use Linux `perf` to record CPU stall cycles and total LLC misses every 20ms. Intel PEBS samples LLC misses at 1-in-100 rate, providing 1% page access samples. We attribute each 20ms stall window proportionally to sampled page accesses (see §4.3 for validation), then average these into per-page stalls (PAC). Figure 1 shows PAC value distributions for each access frequency group using violin plots. Orange lines indicate min, median, and max values.

Masim. We extend Masim, the memory access pattern simulator from Linux's DAMON subsystem [9, 41], to precisely control access frequency and patterns. We use two read-only threads, one for array traversal and the other for pointer-chasing random accesses. Each executes 1.5 billion loads with uniform page access probability. Figure 1a shows a clear bifurcation: PAC clusters around 13 cycles for sequential and 21

cycles for random accesses, despite identical access counts. The gap arises from reduced prefetching and lower MLP in random accesses, while the small absolute PAC values reflect the lack of heavy computation over the data in Masim.

GUPS. We modify GUPS, which performs random updates across large memory regions, to alternate between sequential and random access phases with 50% mix under 1:1 read/write ratio. Figure 1b shows dramatic PAC variation even among pages with identical access counts. For example, pages accessed four times experience stall ranges from under 30 to over 120 cycles, a 4× difference. Higher access frequencies do not necessarily lead to higher PAC values, and the wide spread under uniform frequency further underscores the inadequacy of frequency alone. The higher PAC values compared to Masim stem from GUPS's greater computational intensity, which amplifies the stall cost per access.

tc-twitter. Finally, we evaluate tc-twitter, a real-world graph workload that performs triangle counting over a sparse Twitter graph. This workload exhibits highly complex memory behavior, combining sequential traversals, random accesses, and dependent pointer walks. As shown in Figure 1c, stall costs vary substantially both within and across frequency groups, despite a weak overall correlation between access frequency and stall cost. Compared to previous workloads, tc-twitter incurs significantly higher stall costs due to its complex access patterns and dependencies, which reduce CPU efficiency and expose stalls more frequently. It also exhibits a much wider distribution of PAC values, with single-access pages incurring stalls ranging from as few as 7 cycles to over 460 cycles (65×). In summary,

Takeaway #1: Access frequency fails to capture performance criticality; infrequently accessed pages can incur high stall costs. Pages with the same access frequency can differ in stall cost by up to 65× within a single application. Different access patterns produce distinct stall behaviors, highlighting the need for adaptive tiering that accounts for cross-workload dynamics.

4 PACT Design and Implementation

We present PACT design by first outlining its core principles and challenges (§4.1). We then describe its main components: per-tier stall modeling (§4.2), lightweight PAC profiling (§4.3), an eager demotion policy (§4.4), and an adaptive promotion policy that balances performance and migration overhead (§4.5). Finally, we detail the implementation of PACT (§4.6).

4.1 Goals, Challenges, and Overview

PACT introduces a new principle for tiered memory management: online, performance-criticality-driven page tiering. It prioritizes fast-tier placement based on each page's actual impact on application performance, capturing how much every access contributes to CPU stalls.

Realizing this vision raises three fundamental challenges:

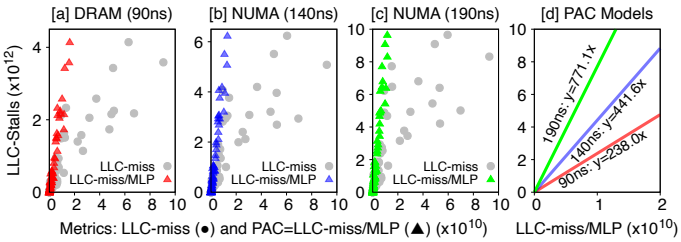

Figure 2. PAC modeling. *PAC can be accurately modeled from latency, LLC misses, and MLP under 96 workloads across three setups.*

(1) How can we quantify PAC to specific tiers at page granularity despite the lack of hardware support? CPU stalls are reported at the processor level, requiring new techniques to attribute per-tier stalls to individual pages. (2) How can we continuously monitor PAC and adapt to dynamic workload phases without incurring prohibitive overhead? (3) How should we design tiering policies for PAC? It requires rethinking when and how to promote, demote, or allocate pages based on performance impact under highly skewed PAC distributions.

PACT addresses these challenges through three key innovations: (1) *Per-tier stall modeling*: We develop the first lightweight mechanism to attribute CPU stalls to individual memory tiers using readily available hardware counters, enabling fine-grained performance tracking without hardware modifications. (2) *Online PAC profiling*: We introduce real-time PAC estimation that eliminates the semantic gap between coarse-grained criticality measures and fine-grained migration decisions. (3) *Criticality-centric policies*: We design PAC-centric adaptive promotion and demotion strategies to balance performance gains with migration overhead.

4.2 From Per-tier Stall to PAC Modeling

Profiling PAC requires two key steps: (a) estimating CPU stalls induced by each memory tier (*i.e.*, per-tier stalls), and (b) attributing those stalls to the responsible pages in a lightweight, online manner. Below we present our in-depth study which reveals key insights to enable per-tier stall estimation as a first step for PAC profiling.

Takeaway #2: Despite the lack of direct hardware support, per-tier CPU stalls can be accurately modeled as a function of LLC misses and per-tier MLP.

$$\text{LLC-stalls} = k \times \frac{\text{LLC-misses}}{\text{MLP}} \quad (1)$$

where k is a per-tier coefficient that captures memory latency and architectural factors. The equation's power lies in its ability to distinguish between memory tiers while accounting for parallelism effects.

We run 96 widely used memory-intensive workloads, such as in-memory caching [11], graph processing [13], ML [5], and HPC [12]. We collect detailed performance counters for offline analysis. Detailed experimental setup is in §5.

Table 1. Hardware counters for PAC sampling.

PEBS	MEM_LOAD_L3_MISS_RETIRED	Slow-tier LLC-miss events/counter
T_1	TOR_OCCUPANCY	TOR queue occupancy
T_2	TOR_OCCUPANCY_COUNTER0	#cycles w/ outstanding TOR entries

We use three memory configurations: local DRAM (90ns), NUMA (140ns), and simulated CXL with CPUless NUMA (190ns). Figure 2 shows the aggregate stalls across different workloads, where each data point represents one workload.

4.2.1 Per-tier Stalls. In Figure 2abc, the gray dots represent the correlation between the number of LLC-misses and LLC-miss-induced stalls (*i.e.*, LLC-stalls), while the triangular markers represent predictions from our MLP-based model (Equation 1). Compared to LLC-misses, the model exhibits a significantly stronger linear relationship with LLC stalls, with Pearson correlation coefficients of 0.98 for the three configurations, respectively, versus 0.82–0.89 for LLC-miss.

The theoretical foundation stems from Little's Law and queuing theory applied to memory subsystems. Each LLC miss incurs a latency cost proportional to the tier's latency; however, modern out-of-order processors can issue multiple concurrent requests, creating overlapping execution windows that amortize latency impact. The MLP factor captures this effect: higher MLP reduces per-access stall contribution.

The coefficient k incorporates tier-specific factors including loaded latency, memory controller queuing delays, and architectural constants. Critically, our extensive validation across 96 workloads and three latency configurations demonstrates that k exhibits strong workload independence while accurately tracking hardware-specific characteristics. This stability stems from MLP's fundamental role in hiding memory latency, a property that remains consistent across diverse access patterns within the same hardware configuration. The formula also generalizes to bandwidth-bound scenarios, where heavy contention inflates effective latency; this inflation is captured by an increased k (the slope of the lines in Figure 2c), reflecting the higher stall cost per access.

4.2.2 Per-tier MLP. Per-tier LLC-misses are directly observable via hardware performance counters. However, there are no dedicated counters for measuring *per-tier* MLP. Existing offcore metrics only capture system-wide parallelism and cannot distinguish between memory tiers. For example, Intel provides Info_Memory_Latency_Load_L2_MLP, which quantifies the average number of L2 cache misses, but offers no tier-specific visibility [7]. What is missing is an equivalent metric for quantifying MLP on a per-tier basis.

To this end, we develop an approach for estimating per-tier MLP, grounded in key architectural observations of how memory requests flow through the memory hierarchy.

Takeaway #3: Per-tier MLP can be measured using CPU CHA queues' occupancy, which reflects the number of concurrent requests serviced by each memory tier.

Modern Intel processors rely on Caching and Home Agent (**CHA**) to coordinate memory traffic between CPU cores and offcore (*e.g.*, DRAM/CXL). When a core experiences an L1/L2 cache miss, the request is forwarded to the CHA for an LLC lookup. If the request also misses in the LLC, the CHA then dispatches it to the corresponding memory tier. Sitting between the CPU cores and offcore, CHA is able to capture all memory requests going offcore, serving as an ideal vantage point for observing per-tier core-to-memory traffic patterns. This architectural structure enables per-tier tracking of outstanding memory requests by observing CHA queue occupancy, offering a low-overhead method to estimate per-tier MLP in real time (more later).

4.2.3 Per-page Access Criticality (PAC). Combining these insights, we can accurately model per-tier stalls, *i.e.*, the performance impact caused by slow-tier page accesses. However, it does not by itself identify which individual pages are responsible for the observed stalls.

Takeaway #4: Per-tier MLP exhibits *periodic stability*, allowing uniform attribution of CPU stalls to individual memory accesses within each sampling window. Accordingly, PAC can be computed as the unit stall cost multiplied by a page's access frequency.

Modern workloads often exhibit *phased* MLP behavior, where MLP remains relatively stable over short intervals (on the order of tens of milliseconds). Within such stable windows, proportional stall attribution based on access frequency provides a reasonable first-order approximation.

Rationale. Workload execution proceeds in phases with quasi-stationary memory access patterns. Within each phase, out-of-order execution and prefetching maintain a nearly constant effective MLP, making individual tier misses approximately exchangeable with a constant per-miss stall cost. Thus, a page's PAC is its miss count multiplied by this unit cost. Streaming phases with high MLP incur low per-miss cost, while pointer-chasing phases with low MLP incur higher cost; bandwidth contention is absorbed as increased effective latency, preserving linear attribution.

Measuring per-tier MLP and LLC-stalls. We use queues in CHA, namely Table Of Requests (**TOR**) to track pending requests. Table 1 shows the counters for PAC estimation. In particular, the per-tier MLP can be calculated (by definition) as: MLP = $\frac{T_1}{T_2}$, where T_1 (TOR_OCCUPANCY) records the number of requests in the TOR, and T_2 (TOR_OCCUPANCY_COUNTER0) counts the number of cycles during which the TOR was not empty. Intuitively, it represents the average number of in-flight requests per active cycle. Figure 3a shows that TOR-MLP (blue) closely matches L2MLP (red), a system-wide metric [7].

MLP periodic stability. Figure 3b presents zoomed-in views of MLP over a 20-second interval, revealing that MLP is inherently phase-oriented: it remains stable at fine temporal granularity and evolves primarily when execution phases

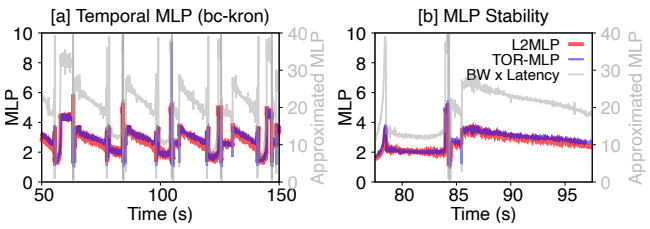

Figure 3. Per-tier MLP. *(a) TOR-MLP accurately tracks temporal MLP trends; (b) Short sampling windows ensure MLP stability within each attribution interval. The gray line shows MLP estimated as* $\frac{Bandwidth \times Latency}{64B}$*, which captures similar trends but overestimates MLP due to aggregate in-flight bytes beyond demand-data.*

change over longer timescales (seconds). Critically, phase shifts occur at larger timescales. This property is key to our uniform stall attribution: within each 20ms window, MLP varies minimally, allowing PACT to treat all sampled accesses as experiencing similar memory-level parallelism. When phases do shift, PACT adapts in the subsequent window by recomputing MLP from fresh counter deltas. This temporal stability reduces noise in PAC estimation and avoids spurious promotions from transient MLP fluctuations.

Portability across hardware. The per-tier stall and MLP-based approach is general enough to extend beyond Intel processors. For instance, AMD's Zen 4 architecture exposes equivalent performance-monitoring facilities required by PACT during its PAC-sampling phase, including LLC misses via IBS (analogous to Intel PEBS) and LLC stalls via PMU events [1, 10]. Unlike Intel, AMD platforms do not expose TOR-like queues for directly measuring per-tier MLP. Instead, Little's Law can be applied: MLP ≈ Latency × Bandwidth [38]. This allows per-tier MLP to be estimated using hardware counters that report average memory latency and bandwidth. The gray line in Figure 3 shows that this MLP approximation closely tracks the temporal trends of the ground-truth MLP, although it overestimates MLP due to accounting for non-demand traffic such as prefetches. Importantly, as shown later, MLP is used to amortize the latency cost of each slow-tier page access in computing PAC; therefore, accurately capturing MLP's temporal variation is more critical than matching its absolute value. Thus, PACT requires only a thin counter-translation layer to operate on AMD platforms.

4.3 PAC Sampling

Algorithm 1 describes how PACT transforms system-level performance metrics into fine-grained PAC estimates through a two-stage sampling and attribution pipeline. The process operates every 20ms using Table 1 counters, making it practical for deployment without specialized hardware support.

4.3.1 PAC Algorithm. The first stage captures aggregate slow-tier performance impact using Equation 1, measuring LLC misses and per-tier MLP. The second stage employs Intel PEBS to sample individual page accesses at configurable

Algorithm 1: PAC Sampling (every 20ms)

```
Measure slow-tier metrics: LLC-misses, MLP ← ΔT1/ΔT2
Estimate slow-tier stalls: S ← k·LLC-misses/MLP
Sample page accesses using PEBS:
    Record per-page (p) info: vaddr, access count (A_p)
Update total accesses: A_t ← Σ_p A_p
foreach sampled page p do
    Attribute stalls to page: S_p ← S · A_p/A_t
    Update PAC: PAC[p] ← α · PAC[p] + S_p        α ∈ [0, 1]
```

rates, enabling scalable per-page tracking. Temporal aggregation accumulates PAC values for each page over time. These accumulated PAC values are later used to guide migration decisions, ensuring that the most performance-critical pages are promoted to the fast-tier.

During each 20ms sampling window, PACT performs two concurrent activities. First, it collects page-granular memory accesses via PEBS, recording for each sampled page p its virtual address (vaddr) and access count (A_p), while accumulating the total sampled accesses A_t across all pages. Second, at both the *start* and *end* of the window, PACT reads the T_1 (`TOR_OCCUPANCY`) and T_2 (`TOR_OCCUPANCY_COUNTER0`) counters. These are cumulative counters, so PACT computes per-tier MLP as $\frac{\Delta T_1}{\Delta T_2}$ using their deltas. This ensures that the resulting MLP value is temporally aligned with the PEBS samples collected within the same window. Using Equation 1, PACT then estimates total slow-tier stalls $S = k \cdot \frac{\text{LLC-misses}}{\text{MLP}}$.

In the attribution phase, PACT proportionally distributes S across all sampled pages according to their relative access frequency. Specifically, for each page p, the attributed stall cycles are computed as $S_p = S \times \frac{A_p}{A_t}$. Lastly, PACT updates PAC using an optional cooling mechanism with factor α to balance long-term criticality with recency. Over time, this sampling and attribution process enables PACT to maintain a fine-grained, dynamic estimation of the true performance criticality of each accessed page.

4.3.2 Validity of Proportional Attribution. Our current PAC sampling relies on proportional attribution, which assumes that within a short execution window (*e.g.*, 20ms), memory accesses belong to a stable execution phase. Under this condition, per-access stall cost becomes a phase-level property, and proportional attribution by access frequency accurately captures each page's relative contribution to CPU stalls. This design has two key advantages. First, PAC attributes stall cost rather than access frequency, directly targeting performance impact rather than locality. Second, by periodically re-estimating PAC, the system naturally tracks long-term shifts in access patterns and MLP behavior, enabling online adaptation to changing phases even when access frequency alone remains stable.

4.3.3 Sampling Period. The choice of sampling period length is critical for accurate per-access stall attribution. Ide-

ally, the sampling window should be long enough to smooth out short-term MLP fluctuations, while short enough to remain responsive to workload phase changes. A sampling period that is too short risks excessive overhead and transient noise, while a period that is too long risks averaging out meaningful variations in page criticality. Given that page migration decisions typically occur at sub-second timescales, PACT samples PAC every tens of milliseconds. By default, PACT uses a 20ms sampling window, as `perf` cannot provide precise counter measurements at sub-10ms scales. This interval offers sufficient resolution to distinguish PAC by capturing MLP dynamics over time, while still maintaining the assumption that MLP remains stable within each window. In practice, we find that 20ms is sufficient to capture PAC dynamics for guiding migrations effectively. §5 shows PAC sampling period can be safely increased to hundreds of ms.

4.3.4 Cooling. Optional temporal decay applies a moving average to PAC values with decay factor $\alpha \in [0, 1]$. This helps improve responsiveness for workloads with time-varying access patterns while avoiding excessive cooling that can induce thrashing. The effectiveness of cooling depends on several interdependent parameters, such as decay period, factor (*e.g.*, EWMA-based), and thresholds, where poorly tuned settings often harm performance more than they help [55].

In PACT, cooling is not a primary design goal. Because PAC values are highly skewed, newly critical pages naturally rise into higher-priority bins without requiring explicit decay. Accordingly, the default configuration (α=1.0) uses pure accumulation, which our evaluation shows to be robust and effective across diverse workloads. This contrasts with conventional hotness-based tiering systems that rely heavily on cooling for performance. We also conduct sensitivity studies on two cooling mechanisms and find their performance impact to be limited (§5). A more systematic exploration of robust cooling mechanisms is left to future work.

4.3.5 Page Access Sampling. We sample only `load` operations, since CXL/NUMA links are full-duplex and write performance impact is minimal. PEBS sampling runs at a default interval of 400 (one sample from every 400 events) and event logs are processed by a dedicated thread. This ensures that PAC is updated promptly after each sampled page access without introducing excessive sampling overhead.

By design, PACT can sample both fast- and slow-tier accesses to guide promotions and demotions. To balance PEBS overhead and accuracy, however, we sample only slow-tier accesses and rely on Linux's (MG)LRU policy for demotion. Coupled with PAC-guided promotions, this proves sufficient for effective tiering (see §4.5). PACT is not bound to PEBS for access sampling; it can readily integrate with other low-overhead access tracking mechanisms. For instance, the recently introduced CXL Hotness Monitoring Unit (CHMU) in CXL 3.2 [3, 46, 59] provides a promising path toward more efficient and accurate memory access sampling.

4.3.6 PAC Tracking. For each accessed page, PACT maintains a compact data structure that records its accumulated PAC value along with a small set of metadata. To ensure that updates are promptly visible to the migration policy, all sampled PAC data is stored in an in-memory hash table, enabling constant-time insertion, lookup, and deletion, critical for managing large volumes of tracked pages.

While hash table supports efficient PAC updates, it does not maintain pages in sorted order, making it difficult to identify highest-impact pages for promotion. Threshold-based classification schemes (*e.g.*, hot/warm/cold) used in prior work [29, 44] are unsuitable here: they rely on heuristic cutoffs that are hard to tune or generalize across workloads, and cannot adapt to the dynamic nature of PAC (§3). Moreover, the wide spread of PAC values introduces heavy skew, further complicating the use of static bins for priority ordering.

To address these challenges, PACT augments its hash table with lightweight priority queues and a dynamic binning mechanism that continuously repositions pages according to their PAC values (§4.5). This design enables efficient, on-demand selection of high-PAC pages for migration.

4.3.7 PAC Limitations and Future Work. The limitations of proportional attribution arise when multiple, heterogeneous memory access patterns are colocated within the same memory tier, such as multi-tenant CXL memory deployments. In such scenarios, latency-bound accesses (*e.g.*, pointer chasing) and latency-tolerant accesses (*e.g.*, streaming) may share a tier but contribute unevenly to observed stalls. Without fine-grained visibility into stall attribution across tenants or access streams, frequency-based proportionality in PAC can dilute the measured criticality of truly latency-sensitive pages. *However, we argue that this limitation reflects an observability gap rather than a flaw in the criticality-first principle itself.* Indeed, even under colocation, the periodic nature of PAC estimation continues to capture phase-level shifts, and our experiments show that PACT still outperforms existing tiering policies under controlled colocation microbenchmarks (Figure 12).

We view our current approach as an initial instantiation of online criticality sampling, analogous to early access-frequency sampling techniques that were later refined through decades of work [18, 29, 39–42, 44, 54, 57, 59]. More accurate PAC estimation under colocation will require finer-grained stall observability, which is out-of-scope for this paper and we leave it as future work.

Encouragingly, recent hardware trends suggest that such observability is becoming increasingly practical. Modern Intel processors (since Sapphire Rapids) extend PEBS with per-load and per-store exposed latency reporting, providing instruction-level visibility into memory access cost [14, 16, 26]. These features align closely with the PAC abstraction and could be directly leveraged to refine stall attribution beyond proportionality. For example, proportional attribution

Algorithm 2: PAC-based Page Promotions and Demotions

Input: Memory pages with accumulated PAC values
Output: Promotion and demotion decisions

```
1 Initialize m                                  ▷ Set demotion aggressiveness
2 while B_priority ≠ ∅ do
3     p ← getPage(B_priority)                   ▷ Pick a candidate page
4     N_promoted ← getPromoted()                ▷ Number of promotions
5     if N_demoted < N_promoted + m then
6         Demote()                              ▷ Make space for promotion
7     Promote(p)
```

can be extended to incorporate sampled per-page latency: $S_p = S \times \frac{A_p \cdot l_p}{\sum_i A_i \cdot l_i}$, where A_p is the number of accesses to page p and l_p is its sampled latency. Such latency-weighted attribution would better separate latency-sensitive and latency-tolerant accesses under colocation while preserving the online, page-granular nature of PAC. We leave the integration of these emerging hardware mechanisms to future work.

4.4 Migration Policies

PACT's tiering policy centers on PAC to guide page migrations, deciding when to migrate, which pages to move, and how often, to balance performance gains against migration overhead. PAC opens up a broad design space for tiering policies, but in PACT it is used primarily to drive page *promotions*. An abstract view of the policy is shown in Algorithm 2. At a high level, PACT's tiering policy consists of two key components: eager demotion and adaptive promotion.

4.4.1 Eager Demotion. PACT's migration policy balances the need to free fast-tier space for high-PAC pages against the risk of unnecessary churn. Instead of waiting for memory pressure, PACT proactively reclaims space from the kernel's LRU list to make room for performance-critical promotions.

At runtime, PACT tracks the cumulative number of promotions and demotions and enforces a balancing rule to preserve adequate fast-tier capacity. Promotions are gated by available space, which is maintained through selective demotion of the least recently used pages. This ensures that fast-tier space is reserved for truly critical pages while avoiding promotion stalls. Early in execution, when fast-tier utilization is low, PACT aggressively demotes inactive pages to build headroom. As utilization grows, it gradually reduces demotion frequency and converges toward traditional on-demand behavior. During each decision window, PACT selects a batch of high-PAC promotion candidates and ensures that at least an equal number of cold pages are demoted, maintaining a balanced promotion–demotion cycle.

PACT also supports a configurable proactive mode (larger m) in which it intentionally demotes more pages than strictly required to create a larger reserve of free fast-tier space. This strategy is particularly effective for workloads with bursty allocation or access patterns, where timely promotion of critical pages might otherwise be blocked.

4.4.2 Migration Algorithm. The overall policy is shown in Algorithm 2, where PACT dynamically adjusts the aggressiveness of demotion. The parameter m specifies how many pages may be proactively demoted (Line 1), with the goal of reserving sufficient fast-tier space for upcoming promotions. To this end, PACT seeks to maintain the number of demoted pages higher than the number of promoted pages. By default, PACT uses $m = 0$ to remain conservative and balance promotion with demotion. However, when the promotion rate is high, increasing m allows additional pages to be demoted in advance, ensuring that fast-tier space is always available. This mechanism provides fine-grained control over the demotion rate. Once a page enters the priority bin B_{priority}, it is promoted immediately. Specifically, PACT continuously monitors the bin (Line 2); when a page becomes available, the *getPage*() function retrieves it (Line 3), pushes it to the promotion queue, and invokes *Promote*(p) (Line 7) to migrate the page using the `move_pages()` syscall. Based on m, if PACT detects that the total number of demoted pages N_{demoted} is less than the total number of promoted pages N_{promoted}, it proactively triggers additional demotions (Line 5–6).

4.5 Adaptive Promotion

While PACT naturally prioritizes pages with highest PAC, determining optimal promotion timing and intensity presents a fundamental challenge: PAC value distributions exhibit extreme workload-dependent skewness and can shift dramatically over time (§3). Unlike prior tiering systems that use fixed thresholds with static binning, PACT must adapt promotion pressure dynamically to vastly different criticality profiles across workloads. Clustering most pages into few bins can cause erratic promotion behavior ranging from fast-tier under-utilization to excessive migration storms.

PACT employs a histogram-based binning approach to capture relative criticality while enabling efficient modulation of promotion intensity. The key innovation lies in its dynamic bin boundary adaptation, which preserves stable promotion behavior despite varying PAC distributions, from uniform spreads in synthetic workloads to extreme power-law distributions in caching and graph analytics applications.

Static binning. To control promotion intensity, PACT initially groups pages into a fixed number of bins (*e.g.*, 20) based on their PAC values. Bin #20 holds the highest-priority pages, with lower bins representing decreasing criticality. At each migration point, PACT promotes pages from the highest non-empty bin. This approach works well under uniform PAC distributions. However, since PAC distribution is skewed, static binning becomes unstable. A narrow clustering of PAC values can result in either too many or too few pages falling into the top bins, leading to erratic promotions.

Adaptive binning. PACT's adaptive binning leverages principled statistical techniques to maintain stable promotion rates while adapting to diverse workload characteristics. The approach combines two lightweight statistical techniques

Algorithm 3: Adaptive Binning for Page Promotions

Input: Memory pages with accumulated PAC values, N_{page}
Output: Near-optimal distribution of pages across bins

Initialize $R_{\text{sampling}}[\,]$ ▷ Fill Reservoir array
foreach *sampled page* p **do**
  $N_{\text{page}} \leftarrow N_{\text{page}} + 1$
  $\text{rnd} \leftarrow \text{rand}()\ \%\ N_{\text{page}}$
  if $rnd < 100$ **then**
    $R_{\text{sampling}}[\text{rnd}] \leftarrow \text{PAC}[p]$ ▷ Update Reservoir
  $\text{Sort}(R_{\text{sampling}}[\,])$
  $\text{Update}(Q_3, Q_1)$ ▷ Update percentiles
  $W = 2 \times \frac{Q_3 - Q_1}{\sqrt[3]{N_{\text{page}}}}$ ▷ Freedman–Diaconis
  if $\frac{N_{page}}{N_c} > T_{scale}$ ▷ N_c: # of promotion candidates
  then
    $W \leftarrow W \times 2$ ▷ Scale up
  else
    $W \leftarrow W/2$ ▷ Scale down

that enable robust online adaptation without requiring expensive full-distribution tracking:

• *Freedman-Diaconis binning*: PACT determines the optimal bin width W using the Freedman-Diaconis rule: $W = 2\times(Q_3-Q_1)/\sqrt[3]{n}$, where Q_1 and Q_3 are the first and third quartiles of the current PAC distribution, and n is the number of tracked pages. This method provides theoretical guarantees on binning quality by minimizing integrated mean squared error while maintaining robustness to outliers through interquartile range normalization. It adaptively balances granularity and robustness to outliers, ensuring meaningful separation of page criticality levels.

• *Reservoir sampling for online adaptation*: The Freedman-Diaconis rule assumes access to the full data distribution, but in practice, PACT must adapt dynamically as workloads evolve. To this end, PACT uses Reservoir sampling to maintain a small, representative set of PAC values at runtime. This approach avoids the overhead of tracking and sorting all page-level PAC values. PACT maintains a fixed-size Reservoir of k samples. The first k page accesses populate the buffer. Each subsequent page has a k/n probability of replacing an existing sample, where n is the total number of observed accesses so far. This ensures uniform sampling without knowing n in advance. The sampled Reservoir enables efficient and low-overhead approximation of the PAC distribution, allowing quartile estimation and adaptive binning in real time.

At runtime, PACT continuously observes PAC values and builds a histogram distribution using the adaptive binning policy. Pages are dynamically assigned to priority bins based on their current PAC values. Using the sample buffer, PACT recomputes bin width dynamically. When the PAC distribution shifts, the Freedman-Diaconis formula yields a new bin size that smooths page promotion pressure. Pages are assigned to bins based on current PAC values, and those in the highest-priority bin become promotion candidates.

In Algorithm 3, adaptive promotion begins by initializing a fixed-size Reservoir array of 100 entries. The first 100 sampled pages are stored directly. For each subsequent page, PACT generates a random number *rnd* in $[0, N_{\text{page}})$,where N_{page} is the number of tracked pages, and replaces a Reservoir entry if rnd < 100, ensuring uniform sampling via Reservoir sampling. The Reservoir is then sorted to compute the first and third quartiles (Q_1, Q_3), which determine the bin width W via the Freedman–Diaconis rule. This allows PACT to adapt bin boundaries dynamically, yielding stable promotion decisions under diverse workloads.

Scaling optimization. While Reservoir sampling tracks distribution trends, skewed or bursty workloads can still lead to bin collapse, where too many pages crowd into high-priority bins. To mitigate this, PACT includes a scaling optimization. PACT dynamically adjusts the bin width based on the current distribution of PAC values. When the ratio $\frac{N_{\text{page}}}{N_c}$, where N_c is the number of promotion candidates, exceeds a predefined threshold T_{scale}, PACT doubles the bin width to spread pages more evenly across bins. When the ratio falls below the threshold, the bin width is halved to restore sensitivity. This symmetric scaling allows PACT to quickly adapt to both increases and decreases in PAC variance. Using the sampled PAC values, PACT maintains a dynamic bin partitioning scheme aimed at producing a smooth, approximately normal distribution of pages across bins. The highest-priority bin is kept small, typically holding only the top 1–5% of pages, ensuring a stable and bounded supply of promotion candidates without causing sudden migration bursts or starvation.

Adaptive binning enables PACT to respond to workload shifts: when PAC variance increases, bins adjust to preserve prioritization sharpness; when pages cluster, bin widths shrink to maintain selection pressure. By dynamically regulating promotion based on workload criticality, PACT maximizes performance gains while avoiding migration overhead.

4.6 Implementation

We implement PACT on Linux 5.15 and incorporate several modules from TPP [37], including fixes to improve the reliability of LRU-based page demotion. To avoid interference from existing OS tiering mechanisms, we disable NUMA hint fault scanning. To enable efficient PAC profiling, we apply two key optimizations. First, we reduce overhead in the PEBS kernel interface by stripping unused fields from PEBS records, which allows us to use a compact 5MB buffer and sample LLC miss events aggressively with minimal runtime cost. Second, we extend the Linux `perf` subsystem to support direct computation of PAC from PMU counters. We further implement a shared-memory buffer to facilitate low-overhead communication between PACT and `perf`, enabling high-frequency sampling and timely PAC updates. PACT incurs minimal overhead, requiring 25 bytes per tracked 4KB page (0.6% memory overhead). The prototype uses two

dedicated threads: one for PEBS processing and one for migrations. Further reductions in CPU overhead are possible with other implementations (*e.g.*, coroutine-based designs).

5 Evaluation

We evaluate PACT against 7 state-of-the-art tiering systems: Soar and Alto [34], Memtis [29], Colloid [51], Nomad [53], TPP [37], and Linux NUMA Balancing Tiering (**NBT**) [2]. First-touch (**NoTier**) results are also included to show the relative effectiveness of tiering. We also perform in-depth analysis to understand the design trade-offs and runtime behavior of PACT. In detail, we seek to answer the following questions:

- How does PACT perform compared to state-of-the-art?
- How effective is PACT in sampling PAC?
- What are the runtime costs and migration characteristics of PACT, and how do they differ from hotness approaches?
- How sensitive is PACT to different design parameters?

5.1 Experimental Platform

Testbed. We evaluate PACT on a dual-socket Intel Skylake server on CloudLab [15]. Each socket contains a 10-core Xeon CPU (2.2GHz) with 96GB DDR4, providing 52GB/s bandwidth and 90ns access latency for local memory access. Cross-socket NUMA accesses exhibit 32GB/s bandwidth and 140ns latency. To emulate CXL memory, we reduce remote NUMA node uncore frequency, increasing access latency to 190ns (2.1× DRAM latency), consistent with measured CXL device characteristics and prior work [30, 32, 37, 51, 53].

Workloads and configurations. We evaluate PACT using a diverse set of memory-intensive applications, including graph analytics [13], GPT-2 inference [5], in-memory databases [11, 49], and HPC workloads from SPEC CPU 2017 [12]. Unless otherwise noted, we run each workload with 8 threads. The memory footprint (RSS) is between 6.6–40GB, with a bandwidth demand of 6–45GB/s. To comprehensively evaluate PACT, we vary tier ratios, page sizes (4KB vs. THP), and memory pressure, and conduct detailed sensitivity analyses. We evaluate multiple fast/slow-tier memory ratios relative to RSS, ranging from 8:1 to 1:1 to 1:8. We focus our in-depth analysis on a few representative workloads (*e.g.*, `bc-kron`) and report results from more workloads later.

Metrics. Performance is primarily measured in terms of runtime. We report normalized slowdown relative to an ideal DRAM-only baseline. Since no tiering policy outperforms DRAM, this offers a fair and consistent point of comparison across workloads and policies. Smaller slowdowns indicate better performance. We also report internal metrics such as the number of page migrations to analyze policy efficiency. Additionally, the gray line in the graphs, labeled CXL represents the slowdown observed when the workload runs entirely on the slow memory tier.

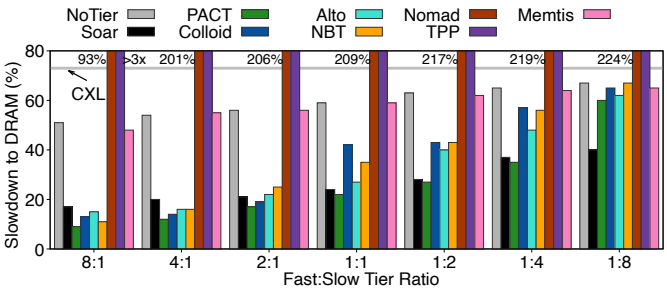

Figure 4. PACT vs. others for bc-kron (4KB page). *PACT is consistently better than Colloid, NBT, Nomad, TPP, and Memtis.*

Table 2. Number of Promotions (bc-kron).

	8:1	4:1	2:1	1:1	1:2	1:4	1:8
PACT	550K	691K	731K	743K	878K	907K	858K
Colloid	1.2M	2M	2.6M	4.7M	4.9M	8M	9M
NBT	1.2M	1.7M	2.6M	3.9M	5.4M	7.4M	8.3M
Alto	812K	956K	1.5M	2.2M	3.7M	5.5M	7.2M
Nomad	32K	19K	17K	19K	9K	13K	5K
TPP	116M	124M	197M	260M	285M	280M	238M
Memtis	4.5K	2.4K	15K	1.3K	1.5K	5K	1.6K

5.2 Graph Workload: bc-kron

We first evaluate PACT using bc-kron, a betweenness centrality approximation workload from GAPBS [13], which captures the challenging characteristics of modern graph analytics. Operating on a synthetic Kronecker graph (134.2M vertices, 2111.6M edges, 20GB footprint), bc-kron exhibits highly irregular, pointer-chasing memory access patterns with minimal spatial locality.

Takeaway #5: Across all fast/slow-tier setups, PACT demonstrates stable and consistent performance, outperforming hotness-based baselines (Colloid, Nomad, NBT, TPP, Memtis) and NoTier in most evaluated ratios, while promoting up to 10.4× fewer pages than the 2nd best for both 4KB page and THP configurations.

4KB pages. Figure 4 compares PACT with state-of-the-art baselines across seven fast/slow tier ratios under 4KB pages. PACT consistently outperforms all baselines while requiring substantially fewer page migrations, confirming our hypothesis that when access frequency is a weak predictor of performance, criticality-driven placement delivers higher performance with lower overhead.

As slow-tier pressure rises, PACT maintains stable performance while competing systems degrade sharply. The NoTier baseline shows only a modest slowdown increase (56% to 62%), whereas hotness-based systems suffer steep declines. For instance, Colloid, the second-best in most cases, sees slowdown grow from 26% to 59% because its aggressive promotion policy triggers frequent NUMA hint faults and migrates large volumes into a constrained fast tier. Under high pressure, its behavior converges toward NoTier, offering little benefit but incurring heavy migration overhead. In contrast, PACT consistently outperforms all baselines,

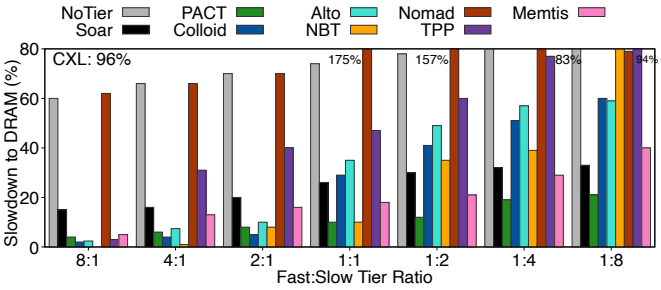

Figure 5. PACT vs. others for bc-kron (THP). *PACT achieves significant performance gains under higher slow-tier ratios.*

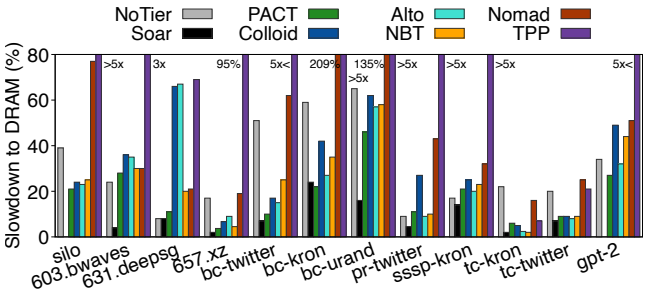

Figure 6. PACT vs. others across 12 workloads. *PACT outperforms Colloid by up to 33% and Nomad by over 500%.*

including NoTier. Nomad performs the worst overall, with slowdowns consistently exceeding 100% due to its reliance on replication across tiers, which increases pressure under heavy migration. TPP fares even worse, with slowdown approaching 800%. At higher slow-tier ratios, PACT continues to scale gracefully, showing modest additional slowdown. Overall, PACT outperforms all baselines by 2–22% while performing up to 2.1–10.4× fewer promotions than Colloid and 1.2–9.6× fewer than NBT.

Understanding PACT benefits. The second-best system, Colloid (and NBT), seeks to balance latency across tiers by exploiting the observation that a heavily loaded fast-tier can sometimes exceed slow-tier latency. Colloid sustains moderate performance through aggressive promotion, but at the cost of massive migration: 11M promotions at 1:1, rising to 30M at 1:8. In contrast, PACT identifies performance-critical pages adaptively via dynamic PAC tracking, sustaining performance with 800K promotions (7× and 20× fewer than Colloid at 1:1 and 1:8, respectively).

Why PACT benefits graph workloads. Despite their "random" appearance, graph workloads exhibit exploitable structure. High-degree vertices (*e.g.*, hub nodes) are accessed frequently and with serialized pointer-chasing, creating low-MLP, high-stall accesses. Frontier-based traversals repeatedly touch the same working set within each BFS/SSSP iteration. PAC naturally identifies these latency-critical pages: even if overall access patterns seem random, PAC captures that some pages consistently cause high stalls due to low MLP. Hotness-based systems treat all frequently accessed pages equally and miss this distinction.

THP. Figure 5 shows that with THP, PACT consistently achieves the lowest slowdowns across nearly all fast/slow-tier ratios. Among baselines, Memtis performs better than Colloid, NBT, and Nomad due to its THP-awareness, yet still lags PACT by 1–19%. Others that fare well with 4KB pages (*e.g.*, Colloid) show higher variance and degraded performance under THP, reflecting their inability to adapt to hugepages. These results confirm that PACT's criticality-driven design generalizes robustly across page sizes, outperforming even THP-aware baselines. For the THP experiments, we enable opportunistic transparent huge pages using `madvise(MADV_HUGEPAGE)`. PACT is inherently robust to mixed page sizes: while PEBS-based sampling reports memory accesses at 4KB granularity and PACT tracks page criticality at this fine granularity, migration decisions are optimized for efficiency. Specifically, when a selected 4KB page belongs to a 2MB huge page, PACT migrates the entire huge page using `move_pages()`, thereby amortizing migration overhead. This design combines fine-grained criticality detection with coarse-grained, cost-efficient memory migration.

5.3 All Workloads

Takeaway #6: Across 12 workloads, PACT maintains robust performance advantage and migrates up to 50.1× and 40.6× fewer pages than Colloid and NBT, respectively.

Figure 6 reports results of 12 workloads under 1:1. PACT consistently outperforms (almost) all hotness-based tiering systems with only marginal losses in the remaining cases. In memory-intensive graph workloads like `bc-urand`, PACT reduces slowdown by 20% over Colloid and 80% over Nomad. For `gpt-2`, where all hotness-based systems perform worse than NoTier due to aggressive yet ineffective migrations, PACT is the only system to outperform it, achieving just 27% slowdown versus 51% and 49% for Colloid and Nomad, respectively. We also evaluate PACT by varying the fast-tier ratio across all workloads. To provide a broader and more comprehensive evaluation, we focus on the representative 1:2 and 2:1 ratios, which capture contrasting memory pressure scenarios and allow us to study the overall behavior of PACT at scale. Figure 7 shows the CDF of PACT performance improvement.

When aggregating results across the three strongest competing systems Colloid, NBT, and Memtis, PACT delivers consistent and robust performance gains. Specifically, PACT achieves an average improvement of 9.95% under 1:2 and 10.66% under 2:1, with peak improvements of 57% and 61%, respectively. Figure 7a shows the distribution of performance improvements over all competing systems under both ratios. The similar distributions indicate that PACT behaves consistently across different tier size asymmetries and scales effectively across a diverse set of workloads. Figure 7b and Figure 7c break down improvements over each individual tiering solution. Under 1:2, PACT achieves average improve-

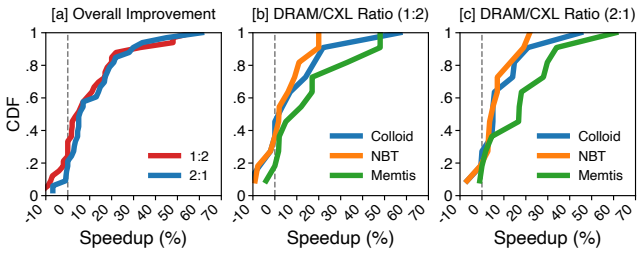

Figure 7. PACT improvement. *Each figure presents the CDF of PACT performance improvement relative to other tiering solutions.*

ments of 9.69% over Colloid, 4.99% over NBT, and 16.6% over Memtis, with maximum gains of 57%, 20%, and 48%, respectively. This trend largely persists under 2:1, further demonstrating the robustness of PACT.

For a small number of workloads, such as `657.xz`, Colloid and NBT outperform PACT by 7% and 9%. Similarly, for `tc-twitter`, Colloid and NBT again achieve lower slowdowns by 11% and 7%. These cases stem from the aggressive promotion policies of Colloid and NBT, which trigger frequent NUMA hint faults and migrate large volumes of pages into a constrained fast tier. Under such conditions, aggressive migration can effectively exploit short-term access recency and better satisfy the working set. Only in one case, `631.deepsjeng`, Memtis outperforms PACT by 4% while performing nearly three times more page promotions.

5.4 Comparison with Soar and Alto

PACT vs. Alto [34]. Alto is a tiering policy that regulates promotion rates based on MLP, whereas PACT adapts continuously by directly measuring PAC. Alto relies on system-wide MLP, while PAC uses per-tier, TOR-based MLP to directly quantify slow-tier stall contributions. As a result, PACT is better suited for workloads with dynamic phases and fine-grained criticality variation. We use Alto on top of Colloid. Across varying fast-tier capacity ratios for `bc-kron` (Figure 4 and Figure 5), PACT outperforms Alto consistently, highlighting the effectiveness of PAC-driven tiering. Figure 6 shows that PACT outperforms Alto on 8 of the 12 evaluated workloads, with an average performance improvement of 7.6%. On the remaining four workloads, where tiering performance is already close to that of DRAM, PACT performs slightly worse, with only marginal differences potentially due to PACT's more aggressive PEBS sampling overhead.

PACT vs. Soar [34]. Soar is a profiling-driven memory allocation policy that prioritizes performance-critical objects in the fast tier. Prior work demonstrates that Soar delivers state-of-the-art performance among memory tiering systems, making it a strong reference point for comparison. Although Soar is not directly comparable due to its reliance on offline profiling, we evaluate both systems on the 10 out of 12 workloads for which Soar profiling and execution are feasible, under identical conditions, as shown in Figure 6. Across six workloads, including `657.xz`, `bc-twitter`, and `tc-twitter`,

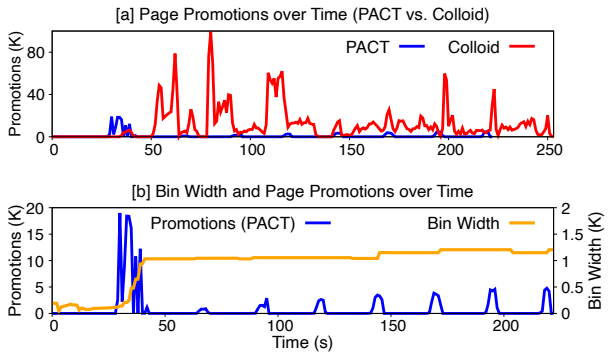

Figure 8. Adaptive page selection. *The figure shows adaptive control of page migration flows in PACT.*

PACT incurs only modest additional slowdown relative to Soar (from 2% to 3.7%, 7% to 10%, and 7% to 9%, respectively), averaging 3.3%. These results indicate that PACT achieves competitive performance despite operating fully online. For `bc-kron`, PACT outperforms Soar by 2% under Soar's static policy. Further analysis shows that `bc-kron` contains a very large object (~16GB) with a high criticality score that cannot be fully placed in DRAM. As a result, the workload benefits less from Soar's object-level placement and does not exploit runtime page promotion to migrate critical pages to DRAM. In contrast, for `603.bwaves`, `bc-urand`, and `sssp-kron`, PACT exhibits higher slowdowns (ranging from 4% to 28%, 16% to 48%, and 14% to 21%, respectively). These cases highlight scenarios where Soar's offline profiling can more effectively capture workload-specific access patterns, illustrating the tradeoff between offline insight and fully online adaptability.

5.5 PACT Adaptivity

Figure 8 illustrates how PACT's adaptive page selection mechanism dynamically regulates promotion activity based on workload behavior in `sssp-kron`. While Colloid triggers over 8M migrations for this workload, PACT performs only 180K, an order of magnitude fewer, yet achieves lower slowdown (18% vs. 25%). Figure 8a shows the resulting effect on the number of promotions over time. PACT's promotion activity initially spikes when PAC variance is high, then quickly stabilizes, with only intermittent promotion bursts thereafter. This behavior demonstrates that PACT can react quickly to changing memory pressure or phase shifts in the workload while avoiding unnecessary migrations. Figure 8b shows how PACT adjusts the bin width used in its dynamic priority binning scheme (§4.5). As the workload progresses, PACT detects shifts in the distribution of PAC values and correspondingly adjusts the bin width to adapt to larger spread in criticality. This adaptivity ensures that the most critical pages are still prioritized, even as the workload evolves. Together, these results highlight how PACT leverages lightweight, runtime-aware profiling to adapt promotion pressure based on PAC, enabling fewer yet more effective migrations that track actual performance impact.

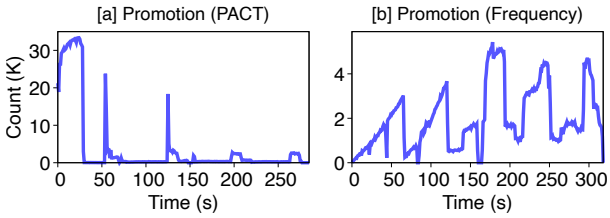

Figure 9. Page promotion (PACT vs. frequency). *The figure contrasts different migration behaviors based on PAC and frequency.*

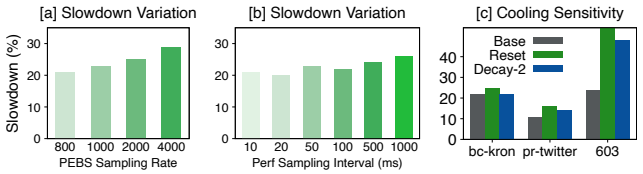

Figure 10. PACT sensitivity analysis. *Slowdown variation under varied PEBS sampling rate and perf sampling interval for bc-kron and Cooling sensitivity comparison.*

5.6 PAC vs. Frequency

To further underscore the distinction between PACT's PAC-driven design and hotness-based heuristics, we implemented a frequency-only policy within the PACT framework. This alternative policy promotes pages solely based on access frequency (part of PAC metadata), mirroring conventional hotness-based tiering strategies. For fair comparison, we configured both policies to perform a comparable number of page promotions. These experiments evaluate whether PAC offers a more accurate reflection of a page's true dynamic performance impact than access frequency alone.

Figure 9 contrasts the promotion behavior under PACT and frequency-based policies. While both systems perform a similar number of promotions overall, their behavior differs markedly. PACT responds promptly to workload demands, front-loading the majority of its promotions in the early phase of execution and then tapering off as fewer high-impact pages remain. In contrast, the frequency-based policy exhibits a periodic, oscillatory pattern, repeatedly ramping up promotions in bursts, suggesting a delayed and less informed response to dynamic workload needs.

This difference stems from the fundamental nature of PAC: it captures per-page performance impact in real time, allowing PACT to quickly surface and promote latency-critical pages. Frequency-based policies, by contrast, react only to accumulated access counts, often missing low-frequency but high-impact pages. This leads to higher LLC stall cycles and degraded performance under hotness-based tiering, despite similar migration counts. Overall, PACT achieves an 18% performance improvement over the frequency-based policy under the same total number of page migrations.

We verified this result generalizes across workloads: under controlled migration counts, PAC-based selection outperforms frequency-based selection by 12–22% across `bc-urand`, `sssp-kron`, and `silo`. The improvement is largest for workloads with high MLP variance, where frequency fails to distinguish between streaming and pointer-chasing accesses.

5.7 Sensitivity Analysis

PACT's performance depends on several key parameters, including the PAC sampling period, PEBS sampling rate, and cooling factors. We conduct an in-depth sensitivity analysis across these parameters to demonstrate PACT's robustness.

PEBS sampling rate. Figure 10a shows the impact of varying the PEBS sampling rate on PACT's performance, measured by slowdown. As the PEBS rate increases from 800 to 4000 (fewer samples collected), we observe a steady increase in slowdown, from approximately 23% to 30%. This trend confirms that denser sampling (lower rate values) leads to more accurate PAC estimation, enabling better page selection and migration decisions. Conversely, sparser sampling limits visibility into per-page access behavior, resulting in degraded performance. This sensitivity study underscores the importance of maintaining a sufficiently fine-grained PEBS sampling rate for effective PAC profiling.

PAC sampling period. Figure 10b shows the impact of varying sampling period from 10ms to 1000ms. At 10ms, we observed a similar number of promotions compared to the default setting, but with a modest 1% increase in slowdown due to the higher overhead from frequent sampling. As the interval increased to 1000ms, we observed a steady increase in both the number of promotions and slowdown. Specifically, promotions rose from 800K (at 20ms) to 1.7M, while slowdown increased from 20% to 27%. This highlights that shorter intervals allow PACT to better track fine-grained fluctuations in memory access criticality, leading to more informed migration decisions. In contrast, longer intervals obscure temporal locality and dilute the distinction between performance-critical and benign pages.

Cooling factor. Figure 10c examines temporal decay factors α. Our default choice of $\alpha = 1.0$ (no cooling) provides robust performance across the evaluation suite, though workload-specific tuning could yield marginal improvements. We implement a lightweight in-place cooling policy. Unlike global methods that rescan all sampled pages to update PAC, our approach tracks, for each page, the number of samples since its last capture. A default threshold of 200K samples (empirically chosen) triggers cooling: when the global sampling counter minus the page's last counter exceeds this distance, its PAC value is cooled. Cooling is applied either by halving the value (decay by 2, *i.e.*, $\alpha = 0.5$) or resetting it to zero ($\alpha = 0$), with the latter emphasizing recency. Figure 10c shows results across three workloads. In most cases, cooling either degraded performance or yielded no benefit, indicating that PACT's online, adaptive policy already responds effectively to workload dynamics without relying on historical PAC values.

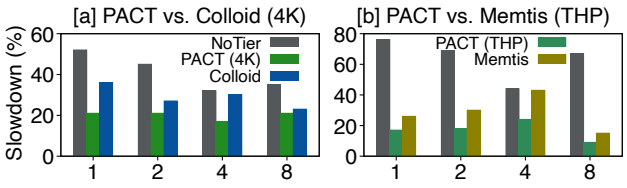

Figure 11. PACT vs. Colloid and Memtis under BW contention. *Constant reduction of page promotion under PACT and PACT with THP compared to Colloid and Memtis for bc-kron.*

Cross-workload robustness. While Figure 10 shows detailed sensitivity for `bc-kron`, we verified that the same trends hold across diverse workloads including `gpt-2`, `603.bwaves`, and `silo`. Across all workloads, default parameters (400 PEBS rate, 20ms period, $\alpha = 1.0$) provide robust performance within 5% of the workload-specific optimum. This consistency stems from PACT's adaptive binning mechanism, which automatically adjusts to workload characteristics without manual tuning. We selected `bc-kron` for detailed presentation because it exhibits the most challenging access patterns (irregular, pointer-chasing) where parameter sensitivity is most pronounced; other workloads show even lower sensitivity.

Takeaway #7: PACT delivers near-optimal performance with default settings and remains robust across a wide range of parameters, eliminating the need for hand-tuning.

5.8 Bandwidth Contention

We next evaluate PACT under *bandwidth contention*, where workloads compete for memory bandwidth.

Setup. We run `bc-kron` while co-locating Intel's Memory Latency Checker (MLC) on the local memory node to simulate background bandwidth-intensive activity. Each MLC thread generates ~8GB/s of traffic, and eight threads fully saturate the DRAM bandwidth. This setup tests whether PACT's PAC-based approach remains effective under bandwidth pressure. For each MLC thread count, slowdowns are normalized to the corresponding DRAM-only baseline under identical contention.

Results. Figure 11 shows performance for PACT, Colloid, and Memtis (for THP) across contention levels (1–8 MLC threads). We choose Colloid and Memtis as they are the second-best performing baselines under 4KB and THP setups, respectively. Across all cases, PACT sustains performance comparable to or better than Colloid and Memtis, while consistently issuing substantially fewer page promotions—3.5–4.7× fewer than Colloid and 2.2× fewer than Memtis. These results demonstrate that PACT maintains high performance with significantly lower migration overhead even under saturated bandwidth conditions.

Takeaway #8: Criticality remains the right signal even under bandwidth saturation: by leveraging per-tier MLP and adaptive binning, PACT continues to guide effective migration decisions despite contention.

5.9 PACT Under Colocated Access Patterns

To validate that uniform stall attribution remains effective when applications with fundamentally different memory behaviors run concurrently, we colocate two `Masim` processes: one with sequential (high-MLP, streaming) access and one with random (low-MLP, pointer-chasing) access. Each process uses a 6GB working set, and we constrain the fast tier to hold only half the total footprint, forcing competition for fast-tier residency.

As shown in Figure 12, PACT reduces slowdown compared to Colloid for both workloads: 112% improvement for the sequential workload, 28% for the random workload, and 61% for aggregate system slowdown. Despite the contrasting access patterns, PACT identifies the dominant source of criticality (the random-access and low-MLP pages) and prioritizes them correctly. PACT achieves this with only 300K promotions versus 12M for Colloid, demonstrating that uniform attribution remains robust even under mixed access patterns. The random workload exhibits higher absolute slowdown as expected (due to inherently serialized accesses), but PACT correctly allocates fast-tier capacity to minimize overall stall time.

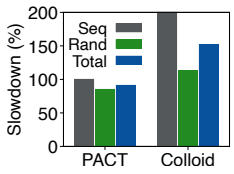

Figure 12. Colocation.

5.10 PACT Breakdown Analysis

We run Redis with YCSB-C to evaluate the contribution of individual PACT techniques. The workload has a 19GB RSS under 1:1. "+Static" uses a fixed bin width for PAC-based page placement, "+Adaptive" dynamically tunes the bin width based on runtime PAC distributions, and "+Both" further applies our scaling optimization to stabilize promotion under skewed or low-variance PAC patterns. Figure 13 shows PACT with "+Both" outperforms Colloid, achieving up to 40% improvements in both latency and throughput while significantly reducing tail latency.

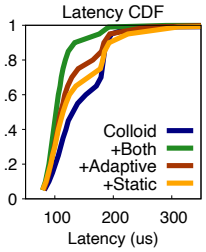

Figure 13. Redis.

6 Conclusion

As hardware-based memory disaggregation and pooling become realities, efficient tiered memory management is more critical than ever. In this work, we establish fine-grained, online performance criticality as a first-class principle for page-level tiering and demonstrate its feasibility through PACT. We believe performance criticality provides a powerful new foundation for rethinking tiered memory management. By introducing PAC and realizing it in PACT, we take an initial step toward criticality-aware memory systems, and hope this work sparks future efforts in building memory hierarchies that are truly performance-centric.

Acknowledgments

We thank Chenxi Wang (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, and Google. Jinshu Liu is supported by a Google PhD Fellowship.

References

[1] AMD64 Architecture Programmer's Manual. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf.

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

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

[4] Giga-updates per second (GUPS). https://en.wikipedia.org/wiki/Giga-updates_per_second.

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

[6] Intel 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[7] Intel PerfMon: Info_Memory_Latency_Load_L2_MLP. https://github.com/intel/perfmon/blob/2336912dd4f4f8f5313914dc30e4f1e87429ab5b/SKX/metrics/skylakex_metrics.json#L12208.

[8] Managing Multiple Sources of Page-Hotness Data. https://lwn.net/Articles/1016722/.

[9] Memory Access Workload Simulator. https://github.com/sjp38/masim.

[10] Performance Monitor Counters for AMD Family 1Ah Model 00h-0Fh Processors. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/58550-0.01.pdf.

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

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

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

[14] Perf Core PMU Support for Sapphire Rapids (Kernel). https://lore.kernel.org/lkml/1611761925-159055-2-git-send-email-kan.liang@linux.intel.com/T/, 2021.

[15] The CloudLab Manual - Hardware. https://docs.cloudlab.us/hardware.html, 2021.

[16] Timed Process Event Based Sampling (TPEBS). https://www.intel.com/content/www/us/en/developer/articles/technical/timed-process-event-based-sampling-tpebs.html, 2025.

[17] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can Far Memory Improve Job Throughput? In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020.

[18] Nadav Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenters. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

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

[26] Roland Kühn, Jan Mühlig, and Jens Teubner. Breaking the Cycle - A Short Overview of Memory-Access Sampling Differences on Modern x86 CPUs. In *21st International Workshop on Data Management on New Hardware (DaMoN)*, 2025.

[27] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[28] 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.

[29] 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.

[30] 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.

[31] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. Utility-Based Hybrid Memory Management. In *International Conference on Cluster Computing (Cluster)*, 2017.

[32] 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.

[33] 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.

[34] 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.

[35] Jinshu Liu, Hanchen Xu, Daniel S. Berger, Marcos K. Aguilera, and Huaicheng Li. Performance Predictability in Heterogeneous Memory. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2026.

[36] 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.

[37] 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.

[38] John McCalpin. Single-core Memory Bandwidth: Latency, Bandwidth, and Concurrency. https://sites.utexas.edu/jdm4372/2025/02/17/single-core-memory-bandwidth-latency-bandwidth-and-concurrency/.

[39] Alan Nair, Sandeep Kumar, Aravinda Prasad, Ying Huang, Andy Rudoff, and Sreenivas Subramoney. Telescope: Telemetry for Gargantuan Memory Footprint Applications. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC)*, 2024.

[40] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. DAOS: Data Access-aware Operating System. In *Proceedings of the 31st IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2022.

[41] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference (Middleware)*, 2019.

[42] 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.

[43] Samir Rajadnya and Durgesh Srivastava. CMS: Hotness Tracking Requirements. https://www.opencompute.org/documents/ocp-cms-hotness-tracking-requirements-white-paper-pdf-1.

[44] 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.

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

[46] 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. 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)*, 2025.

[47] 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.

[48] 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.

[49] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[50] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th International Middleware Conference (Middleware)*, 2015.

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

[53] 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.

[54] Dong Xu, Junhee Ryu, Jinho Baek, Kwangsik Shin, Pengfei Su, and Dong Li. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC)*, 2024.

[55] Sujay Yadalam, Konstantinos Kanellis, Michael Swift, and Shivaram Venkataraman. ARMS: Adaptive and Robust Memory Tiering System. https://arxiv.org/abs/2508.04417.

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

[57] Anil Yelam, Kan Wu, Zhiyuan Guo, Suli Yang, Rajath Shashidhara, Wei Xu, Stanko Novakovic, Alex C. Snoeren, and Kimberly Keeton. PageFlex: Flexible and Efficient User-space Delegation of Linux Paging Policies with eBPF. In *Proceedings of the 2025 USENIX Annual Technical Conference (ATC)*, 2025.

[58] 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.

[59] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Jie Zhang, Yongqiang Xiong, and Guangyu Sun. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering. In *57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-57)*, 2024.