

TIERSizer: Counterfactual Reasoning for DRAM Sizing in Tiered Memory Systems

Sujay Yadalam
The University of Texas at Austin
Austin, Texas, USA

Saarth Deshpande
The University of Texas at Austin
Austin, Texas, USA

Divyanshu Saxena
The University of Texas at Austin
Austin, Texas, USA

Venkat Arun
The University of Texas at Austin
Austin, Texas, USA

Neeraja J. Yadwadkar
The University of Texas at Austin
Austin, Texas, USA

Aditya Akella
The University of Texas at Austin
Austin, Texas, USA

Abstract

Compute Express Link (CXL) enables cost-effective memory capacity expansion by placing additional tiers behind a coherent fabric. To overcome the performance loss induced by the higher latency of CXL, tiering systems keep hot data in DRAM and demote cold data to the slower tiers. While most research has focused on the data placement problem, very little addresses the basic operational question: how much DRAM does an application actually need to meet a performance target on a tiered system? Too much DRAM wastes the savings that motivated tiering; too little degrades performance. The right size is workload-dependent and drifts over time, so operators today over-provision conservatively or accept silent slowdowns.

We observe that sizing DRAM requires predicting an application's slowdown at different DRAM sizes — a counterfactual that must be answered while the workload runs, without instantiating the size. We show this counterfactual is answerable online from commodity system metrics using two simple analytical models, one for the change in LLC-miss stalls and one for migration overhead. Both models account for a real, imperfect tiering engine rather than an idealized one. We build TIERSizer, an online predictor and controller that steps DRAM toward its minimum, in small increments, keeping performance within a target and avoiding tail-latency spikes. On a diverse set of workloads atop a state-of-the-art tiering system, TIERSizer saves 35% DRAM on average (up to 90%) over a 2:1 configuration while staying within 5% of baseline performance.

CCS Concepts: • **Software and its engineering** → **Memory management**; • **Computer systems organization** → **Cloud computing**; • **Hardware** → **Memory and dense storage**; • **General and reference** → Estimation.

Keywords: Memory Tiering, Compute Express Link (CXL), DRAM sizing, Counterfactual Prediction, Performance Modeling, Resource Provisioning

ACM Reference Format:

Sujay Yadalam, Saarth Deshpande, Divyanshu Saxena, Venkat Arun, Neeraja J. Yadwadkar, and Aditya Akella. 2026. TIERSizer: Counterfactual Reasoning for DRAM Sizing in Tiered Memory Systems. In *4th Workshop on Disruptive Memory Systems (DIMES '26)*, September 29–October 02, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3831664.3837791>

1 Introduction

Main memory is a significant contributor to server cost in modern datacenters, and DRAM scaling has stagnated even as application working sets continue to grow [20, 23, 24, 34]. Compute Express Link (CXL) [6] has emerged as a practical response: by attaching additional capacity behind a coherent fabric, operators can extend a server's effective memory capacity at possibly lower cost per gigabyte than adding equivalent DRAM. Accessing data on these new CXL tiers incurs higher latency [21]. As a result, applications experience performance degradation if they repeatedly access data on CXL. Recent research has tackled this by designing tiering systems [7, 11, 15, 18, 22, 29, 30, 36, 38] that aim to smartly place hot data in DRAM and cold data on CXL to minimize the performance penalty.

A central operational question, however, remains largely unaddressed. Given a workload running on a tiered system, *what is the minimum DRAM required to achieve a target performance?* The challenge in answering this question is that the tradeoff between DRAM size and application performance is sharp: over-provisioning DRAM generally preserves target performance but erodes the cost savings that motivated tiering in the first place. On the other hand, provisioning too little pushes hot accesses to slower CXL memory and degrades performance. Figure 1 shows how application performance falls as the DRAM ratio shrinks, with AutoNUMA [7]. The right size depends not only on the application and tiering engine but is also dynamic since working sets shift across workload phases and traffic patterns drift over time. Lacking a principled way to size DRAM, operators today either over-provision conservatively or accept silent performance loss.



This work is licensed under a Creative Commons Attribution 4.0 International License.

DIMES '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2880-8/2026/09

<https://doi.org/10.1145/3831664.3837791>

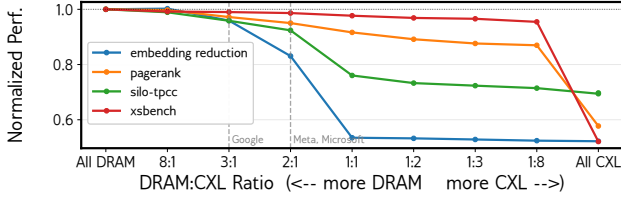


Figure 1. Application performance, when running with AutoNUMA [7], as the DRAM:CXL capacity ratio changes from all-DRAM (fast) to all-CXL (slow), normalized to the all-DRAM configuration. Google, Meta and Microsoft report using 3:1 or 2:1 configurations [11, 23, 39].

For instance, Google, Meta, and Microsoft report DRAM:CXL capacity ratios of 3:1 or 2:1 to ensure minimal performance disruption [11, 14, 23, 39].

Despite its importance, very little work has addressed this question. Answering it ideally requires the trade-off curve for performance slowdown as a function of tier sizes for each workload. A strawman approach is to profile workloads offline but given the diversity and rate of evolution of production workloads, this is expensive, often infeasible, and the profiles may become stale as workloads evolve. Alternatively, a naïve online approach may periodically change the DRAM size and measure the performance; and then choose the smallest size where we meet the target performance. However, blind exploration can be dangerous: even modest reductions in DRAM size can inflict severe slowdowns and SLO violations before the system can react. In Figure 1, for instance, the performance for embedding-reduction degrades sharply as the ratio moves from 2:1 to 1:1.

To avoid such catastrophic choices, what we need is a mechanism that uses system measurables, while an application runs, to answer the counterfactual: *what would this application’s slowdown be at a DRAM size different from what it is currently running on*. However, performing this counterfactual analysis on what an application’s performance would look like at a DRAM size without instantiating it is challenging. This is because estimating the effect of DRAM change is neither uniform nor easy to predict. Shrinking DRAM by ΔD (going from D to D') spills some pages to CXL, but how much additional traffic reaches CXL depends on the workload’s per-page access distribution, *and* on how accurately the tiering system migrates pages between the tiers. That traffic translates into latency, and the added latency degrades performance only to the extent it is not hidden by the application’s memory-level parallelism (MLP) and architectural optimizations such as prefetching. Each of these factors is workload-, hardware-, and time-dependent, so no fixed ratio or one-time profile can capture them.

Our key insight is that the slowdown from a size change in a tiered system has two significant sources: (i) change in CPU stalls caused due to Last Level Cache (LLC) misses [15, 22], which we refer to as LLC-stalls or s_{LLC} , and (ii) change in

overhead due to page migrations. This decomposition lets us answer the above counterfactual with *simple* analytical models using metrics readily available on commodity servers. Simplicity matters for robust extrapolation since it prevents overfitting. It also reduces sample complexity for learning, enabling rapid adaptation to changes in the workload.

Shrinking DRAM pushes the coldest DRAM pages to CXL, and the displaced accesses incur extra latency, but only the fraction that does not hide via memory-level parallelism (MLP) surfaces as additional stalls. Predicting stalls, therefore, needs just three quantities: how many accesses move to CXL, how much slower those accesses will be, and how much of that latency results in CPU stalls. Crucially, we do not assume the tiering engine is ideal; we model its inaccuracy explicitly, since a real engine misplaces pages near the hot/cold boundary and sends more traffic to CXL than an idealized prediction expects. The second source of slowdown is page migrations. As DRAM shrinks, tiering engines migrate more often, and a migrating page occasionally blocks an application thread that faults on it mid-migration. These blocking cycles never surface as LLC-miss stalls, so we model them separately and add them to the prediction. Migrations also contend for bandwidth, but that effect is folded into the latency term of the stall model.

Using these models, we build TIERSizer, an online slowdown predictor and controller that minimizes the DRAM allocated to an application on a tiered system while meeting a target performance. Rather than jumping to the best size directly, TIERSizer steps toward the target size in small increments. This is done for two reasons. First, a large jump risks a spike in tail latency and a burst of SLO violations if the new size turns out to be unsuitable. Second, small steps make the counterfactual trustworthy: between adjacent sizes, an application’s MLP and bandwidth utilization change only slightly, which is precisely the regime our model assumes. Incremental stepping thus keeps the controller within the region where its own prediction holds.

TIERSizer also monitors the tier access ratio, bandwidth utilization, and LLC-stalls to detect workload shifts, phase changes, and environmental changes such as interference from co-located applications, restoring the DRAM size when needed while always keeping performance within the specified limit. We evaluate TIERSizer on top of ARMS [36], a state-of-the-art tiering system, across a diverse set of workloads. TIERSizer saves about 35% DRAM on average, relative to a naïve 2:1 configuration while keeping performance within 5% of the baseline.

In summary, this paper makes the following contributions:

- We demonstrate that application slowdown on tiered memory systems can be predicted online and accurately from readily available system metrics using two simple analytical models: one for LLC-stalls and one for migration overhead.

- We use these models with TIERSizer which uses counterfactual reasoning to minimize DRAM usage while achieving performance within a specified target.
- Our evaluation on diverse set of workloads reveals that TIERSizer can achieve up to 90% DRAM savings while staying within 5% of the baseline performance.

2 Background

Memory tiering. CXL enables cost-effective memory expansion by adding a second memory tier below local DRAM. A CXL access typically incurs 2–3× the local DRAM latency and offers lower bandwidth [21]. Applications see performance degradation if they access data on CXL frequently.

Tiering systems [2, 7–9, 11, 15, 18, 22, 23, 29, 30, 33, 35–38] manage this placement. They estimate page hotness, keep frequently accessed hot pages in DRAM, demote cold pages to CXL, and migrate pages as patterns change. These systems rely on hardware event sampling such as Intel PEBS [16], periodical scanning of page tables [25] or page fault hints to determine page accesses. These approaches yield only an *approximate, noisy* estimate of how hot a page is, which affects tiering quality [17], ultimately affecting application performance. Moreover, these systems tackle only data placement, not DRAM sizing—a problem at least as important as the placement itself.

Performance with varying DRAM sizes. An application’s performance depends heavily on how much of its working set sits in DRAM versus CXL. Figure 1 shows application performance as we sweep the DRAM:CXL capacity ratio from all-DRAM (fast) to all-CXL (slow), normalized to the all-DRAM case. We make two observations. First, performance is less sensitive to shrinking DRAM up to a point, beyond which it drops sharply. Second, the location and steepness of this knee vary widely across workloads. For example, XSBench holds near-DRAM performance down to 1:8 ratio before falling off, whereas Embedding-Reduction(ER) and Silo start degrading as early as 2:1. No single DRAM:CXL ratio suits every workload: a fixed 2:1 over-provisions DRAM for some applications while under-serving others. Further, the amount of DRAM required can vary over time as workloads drift or phase changes. This motivates sizing DRAM *per workload*, and adapting it as behavior shifts over time.

Modeling impact of memory bandwidth and latency on performance. Prior work [5, 28] has built analytical models that quantify how memory latency and bandwidth affect application performance, combining application-level metrics from hardware counters (CPI, MPKI, MLP) with profiled bandwidth-latency curves for the memory system. Clapp et al. [5] use such a model to classify big-data workloads as either bandwidth-bound or latency-sensitive, while PROFET [28] uses it to predict an application’s performance on a memory configuration different from the baseline it was profiled on. We share their central intuition that slowdown is governed

by the increase in CPU stall caused by higher memory access latency and reuse it in our stall model (§3.1). However, both assume a *uniform* memory system, where a configuration change shifts the latency of every access alike. Tiering breaks this assumption: resizing DRAM leaves some access latencies largely unchanged and instead changes *which* accesses are served by the slow tier, a shift that depends on the application’s page-level access distribution and on how accurately the tiering engine places those pages. Second, these approaches are offline: they characterize an application across memory configurations to build a model, whereas DRAM sizing needs a prediction *while* the workload runs, at a size it is not running on. TIERSizer therefore models the change in per-tier access counts under an imperfect tiering engine, adds migration overhead explicitly, and fits its model online.

Predicting and sizing memory. A few systems touch the tiering sizing problem but none answer it directly. AOL [22] predicts an application’s slowdown, but only for a fixed placement, when all of its data sits on CXL, rather than at intermediate sizes. Vulcan [31] divides a fixed pool of DRAM among co-running workloads under current conditions, but does not reason about how much DRAM a single workload needs to meet its performance target. Tuna [4] sizes DRAM from offline benchmarking, matching a running application to pre-profiled microbenchmarks to pick a size; it is static and generalizes poorly to unseen workloads.

3 Estimating slowdown on tiered systems

Selecting the right DRAM size online requires reasoning about how an application’s performance changes across sizes. That, in turn, depends on how the application accesses its pages and which of those pages remain in DRAM versus CXL *under the new size*. We answer this with a counterfactual performance model that predicts an application’s slowdown at a DRAM size it is not running on. The model rests on a simple decomposition of where that slowdown comes from.

An application slows down on a tiered system for two reasons. First, demand loads that miss in the LLC and are served from CXL incur higher latency, increasing CPU stalls [15, 22]. Past work [22] has found that performance degradation on the tiering systems is predominantly caused by such CPU stalls. Second, page migrations triggered by the tiering engine occasionally block the application on a fault when its access races with an ongoing migration (O_{mig}); these cycles are lost to the application but do not appear as memory stalls. The slowdown in an application’s performance $\Delta P_{D \rightarrow D'}$ when moving from a DRAM size D to D' combines the two, normalized by the total cycles in the period:

$$\Delta P_{D \rightarrow D'} = \frac{\Delta stalls + \Delta O_{mig}}{cycles} \quad (1)$$

where $\Delta stalls = stalls(D') - stalls(D)$ is the change in the CPU LLC miss stalls, and $\Delta O_{mig} = O_{mig}(D') - O_{mig}(D)$ is the change in the migration overhead.

Treating the $\Delta \text{stalls}/\text{cycles}$ as the slowdown is known to *overestimate* the true slowdown for high-MLP workloads [22]. We adopt it deliberately: overestimating keeps us conservative, so we never under-predict slowdown and risk violating the performance target. We follow this principle throughout the design. Wherever a modeling choice could err, we bias it to over-predict slowdown, so errors cost slightly more DRAM rather than risk a target violation.

3.1 Estimating LLC-stalls

A demand load that misses in the cache stalls the CPU while it waits for data from memory. The number of stall cycles is proportional to the access latency— L_D for DRAM, L_C for CXL—but not necessarily equal to it. The main factor relating the two is memory-level parallelism (MLP) [22], the number of concurrent outstanding requests to memory. A high-MLP workload issues many accesses in parallel and overlaps their latencies, hiding most of the stall cycles, whereas a low-MLP workload issues requests sequentially and exposes them as stalls. Architectural features such as prefetching, cache replacement, out-of-order execution, and speculation further modulates this relationship. In short, a DRAM access adds at most L_D stalls while a CXL access can incur higher stalls up to L_C , with MLP and the micro-architecture determining how much of that is actually exposed.

Based on this principle, we estimate stalls in a period as

$$\text{stalls} = \alpha_D * \frac{A_D L_D}{MLP} + \alpha_C * \frac{A_C L_C}{MLP} + \beta \pm \epsilon \quad (2)$$

where A_D, A_C are the numbers of accesses served by DRAM and CXL. Each term $\frac{A_t L_t}{MLP}$ is the tier- t access latency left after concurrent misses overlap. The per-tier factor α_t scales it to the fraction that actually surfaces as stalls. We keep α_D and α_C separate because MLP is an aggregate count: it tells us how many requests are outstanding overall, but not how DRAM and CXL accesses overlap among themselves or how much of each tier's latency is hidden. Rather than try to estimate this split directly, we let α_D and α_C absorb it, fitting each to the fraction of its tier's latency that is exposed as stalls. The offset β accounts for stalls from pages the tiering engine does not manage such as code and stack pages, which are independent of the DRAM size. ϵ bounds the residual error.

We obtain A_D, A_C from hardware counters, estimate L_D, L_C from offline bandwidth-latency curves indexed by the measured bandwidth, and MLP from the occupancy of CHA (Cache/Home Agent) queues similar to prior work [22]. We then calibrate $\alpha_D, \alpha_C, \beta$ online by fitting Eq. 2 to measured stalls with a linear-programming solver, refitting each period so the model tracks the current operating point.

Sources of error. We deliberately choose a simple model: it is generic, interpretable, cheap to refit, and because sizing needs only the *difference* in stalls, it remains useful even when its absolute estimates are not perfect. This simplicity, however, introduces errors from three sources. First, we use average

values of L_C, L_D and MLP over a period, which hides variation within it. This is acceptable because total stalls are a sum over every access in the period and so depend on the average cost of an access, not its tail. Sampling more finely (every few ms) would shrink this error but adds overhead, and we find that 1-second averages already give high accuracy. Second, the latencies L_C and L_D are estimated using bandwidth-latency curves measured offline. This ignores variation in latencies due to differences in read:write ratio and access size. We index the curves conservatively and fold the remainder into ϵ . Third, ϵ absorbs the residual, such as variation in workload behavior across periods, slight interference from co-located applications, and noise in the performance counters.

3.2 Modeling migration overhead

While CXL stalls are the dominant source of slowdown, applications on tiered memory systems also incur overhead from page migrations. As DRAM shrinks, the number of migrations could increase, resulting in higher overheads. This overhead takes two forms. The first is bandwidth contention: migrations compete with the application for memory bandwidth and inflate its access latencies. We do not model this separately, as it can be captured by the bandwidth-dependent latency term of our stall model (Equation 2). The second is blocking: when the tiering policy migrates a page, an application thread that touches that page faults and blocks until the migration completes, taking cycles away from useful work. These wasted cycles are not captured by stalls, so we model it explicitly. We call these events *blocking faults*, and only a small fraction of all migrations cause them.

We model the blocking-fault overhead as

$$O_{mig} = N_{BF} \cdot \bar{t}_{block}, \quad (3)$$

where N_{BF} is the number of blocking faults in a period and $\bar{t}_{block}$ is the average time a thread is blocked per fault. We count N_{BF} directly from the page-fault path during the period. To estimate $\bar{t}_{block}$, note that a blocking fault stalls a thread only until the migration of the accessed page completes. Because the access is equally likely to arrive at any point during that migration, the time the thread is blocked is uniformly distributed between 0 and the migration latency t_{mig} , so the average blocking time is half the migration latency, i.e., $\bar{t}_{block} = t_{mig}/2$.

Modeling the overhead of blocking faults helps avoid DRAM undersizing. At small DRAM sizes, the hot set does not fit in DRAM, which results in pages thrashing between tiers. As a result, blocking faults go up raising the predicted migration overhead, and stops the system from shrinking DRAM further.

4 Counterfactual Prediction

The two models in the previous section estimate an application's stalls and migration overhead from quantities measured at its *current* DRAM size. To select the right DRAM size, we must evaluate them at a counterfactual different size D' , which requires estimating their inputs at D' without ever running

the application there. In this section, we describe our system called TIERSizer which uses the models to predict application slowdown at different DRAM sizes.

TIERSizer does not predict slowdown at every candidate size and jump to the best. Instead, it evaluates sizes near the current allocation and takes a small step (10%) in the desirable direction. This is both safer and more accurate. First, a large jump risks overshooting a performance cliff, whereas a misjudged small step is easily recoverable. Second, the model is most accurate around the current operating point. All the model’s inputs, including the access counts, latencies, MLP, and fitted parameters hold only at nearby DRAM sizes. Small steps keep predictions inside this region of validity.

Small steps also simplify the prediction itself. Between adjacent sizes, MLP and the per-tier latencies change negligibly, and the parameters $\alpha_D, \alpha_C, \beta$ fit at the current size remain valid. We therefore hold these at their measured values and re-estimate only what the resize changes directly: the per-tier access counts for the stall model, and the slow-tier access count for the overhead model. We describe each in turn.

4.1 Estimating stalls at D'

Estimating access counts. We track per-page access counts, sort pages by count, and take the coldest ΔD resident pages, those that an ideal engine would demote when DRAM shrinks by ΔD . Their total accesses, ΔA , is the traffic that moves from DRAM to CXL at D' , so $A'_D = A_D - \Delta A$ and $A'_C = A_C + \Delta A$.

This ideal estimate can err in two ways. First, page hotness drifts within a period, so the engine migrates some pages back and forth; the true DRAM traffic is then higher, and CXL traffic lower, than predicted. This direction is harmless as long as the increased DRAM traffic does not inflate DRAM latency substantially. We assume a bandwidth-aware tiering engine such as Colloid [33] or ARMS [36], which monitors per-tier load and balances placement so that neither tier is driven into saturation, keeping DRAM in the flat region of its bandwidth-latency curve. Note that our predicted L'_D is indexed by the *predicted* DRAM bandwidth, so does not account for this extra traffic; ϵ bounds the residual. The net effect nonetheless remains conservative: each retained access avoids a CXL latency that is $2-3\times$ higher, so the reduction in the CXL stall term outweighs the modest increase in the DRAM term.

Second, a real engine does not demote exactly the coldest pages: it relies on noisy, incomplete hotness signals (PEBS sampling, plus approximate bookkeeping), sending more traffic to CXL than ideal. Because this error is systematic and grows with ΔD , we model it directly by correcting ΔA , rather than letting ϵ absorb it.

Insight 1: Mis-placements are higher for warm pages near the hot/cold boundary than extremely hot and cold pages. Pages far hotter or colder than the cutoff are placed correctly; only pages near the boundary are easily confused, due to sampling noise. Let h_i be page i ’s true access count and h^* that of the ΔD -th coldest resident page (the new boundary).

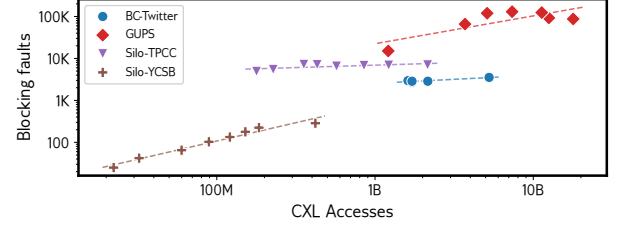


Figure 2. Number of blocking faults N_{BF} versus CXL access count. Data obtained at different DRAM sizes. We omit workloads with very few blocking faults.

Under PEBS sampling rate p , the engine’s estimate of h_i has mean h_i and standard deviation $\sqrt{h_i/p}$ (Poisson sampling), so page i is demoted with probability

$$P_{\text{dem}}(i) = \Phi(z_i), \quad z_i = \frac{h^* - h_i}{\sqrt{h_i/p}}, \quad (4)$$

where Φ is the standard normal CDF and z_i is the page’s distance from the boundary in units of its own noise. Weighting each resident page’s accesses by this probability gives the expected demoted traffic, which replaces the ideal sum:

$$\Delta A = \sum_{i \in \mathcal{D}} h_i \Phi(z_i), \quad (5)$$

with the boundary chosen so that $\sum_{i \in \mathcal{D}} \Phi(z_i) = \Delta D$, keeping the number of demotions equal to the capacity reduction. Intuitively, pages far hotter or colder than the boundary are weighted near 1 or near 0 and are placed correctly. Only the pages near the DRAM boundary, are prone to misplacement, and contribute fractional, error-prone traffic.

If the engine tracks hotness by page-table scanning instead of PEBS, its estimates are coarser; we could model this as a lower effective rate p/κ ($\kappa > 1$), which widens the noise and yields a more conservative correction.

Note that we apply this adjustment only when shrinking DRAM; when growing it, we assume ideal promotion. This asymmetry follows the conservatism principle mentioned earlier: over-predicting the benefit of extra DRAM means we grow readily and stop shrinking sooner, ensuring we meet the performance target at the cost of slightly higher DRAM usage.

Estimating latencies and MLP. Using the estimated access count A'_D and A'_C , along with the migration rate, we predict the per-tier bandwidths, subsequently their latencies L'_D and L'_C at the counterfactual DRAM size. We reuse the currently measured MLP since we find that MLP remains fairly constant at nearby DRAM sizes.

4.2 Estimating migration overhead at D'

From Section 3.2, the migration overhead depends on two values N_{BF} and t_{mig} , so predicting the migration overhead $O_{\text{mig}}(D')$ at D' reduces to predicting the number of blocking faults $N_{BF}(D')$.

Insight 2: Blocking faults scale with slow-tier accesses. Pages in CXL are promoted when they become hot, i.e., receive

many CXL accesses, while cold pages in DRAM are demoted to make room for promotion candidates. Every migration is therefore ultimately triggered by a slow-tier access, so more CXL accesses increase the number of migrations, and in turn the chances to block. Note that demotions are unlikely to result in blocking faults, since the pages chosen are cold and it is unlikely for a thread touch one mid-migration.

This, in turn, increases the chances to block. As shown in Figure 2, we find that N_{BF} is proportional to the CXL access count. We can therefore estimate number of blocking faults at D' as

$$N_{BF}(D') = N_{BF}(D) \left(\frac{A'_C}{A_C} \right)^\eta \quad (6)$$

where the exponent η sets how steeply blocking faults grow with CXL accesses. Rather than fit η per workload, we fix $\eta = 1$, making N_{BF} grow linearly with CXL accesses. Where the true growth is sublinear, this overestimates N_{BF} at smaller sizes, keeping our slowdown prediction conservative, and it spares us any per-workload calibration.

5 Implementation

We integrate TIERSizer with ARMS [36], a state-of-the-art tiering system, though our techniques are not specific to it. ARMS uses PEBS [16] to sample per-page accesses and compute a hotness score for each page. Periodically, it ranks pages by score and migrates top-ranked pages to DRAM and bottom-ranked pages to CXL. TIERSizer adds an online slowdown predictor and a sizing controller on top: it monitors the application, fits the stall model, and adjusts the DRAM capacity.

Monitoring. TIERSizer collects model's inputs from hardware counters once per second. It reads access counts (A_D, A_C), memory bandwidth, LLC-miss stall cycles (s_{LLC}), and derives MLP from the occupancy of the CHA queues [22]. The per-page access counts, number of blocking faults and average migration time, needed for the counterfactual come from ARMS.

The per-tier bandwidth–latency curves are measured once per platform with a memory-load microbenchmark that sweeps offered bandwidth and records the resulting loaded latency, similar to MESS [12]. At runtime TIERSizer indexes them by the measured bandwidth, to estimate the latency.

Fitting the stall model. Every 30 seconds, TIERSizer fits the stall model (Eq. 2) to the measured stalls. We formulate the fit as a linear program, solved with PuLP [27] and the CBC backend. The solver is lightweight and runs off the application's critical path.

Sizing controller. Using the fitted parameters, TIERSizer evaluates the counterfactual slowdown (§4) at neighboring sizes, $D - \Delta D$ and $D + \Delta D$ every period. If the predicted slowdown stays within a target, TIERSizer shrinks DRAM by ΔD ; otherwise it holds, and it grows DRAM if the measured slowdown already exceeds the budget. The step ΔD is small so that MLP and bandwidth stay within the model's valid range.

Workload	Inputs	RSS	Description
Liblinear [13]	kddb	34.13	Logistic regression-based classifier.
GapBS-BC [3]	twitter	13.08	Compute the measure of centrality
GapBS-PR [3]	twitter	12.32	Compute PageRank score of a graph.
GapBS-CC [3]	kroner	69.29	Compute connected components of graph
Silo	TPC-C	75.68	In-memory transactional database.
	YCSB-C	71.40	
Btree [1]	-	12.13	In-memory index lookup benchmark.
XSbench [32]	-	64.97	Compute kernel of MCNP algorithm.
GUPS [26]	8 GiB hot	64.03	Skewed accesses with dynamic hotset.

Table 1. Workloads Characteristics. RSS is in GiB.

Detecting workload and environment changes. TIERSizer watches the per-tier access ratio, bandwidth, and s_{LLC} for abrupt shifts that signal a phase change, workload drift, or interference from a co-located application. Such a shift invalidates the current fit, so TIERSizer refits immediately and, if performance has crossed the budget, reverts to the last known-safe size before resuming.

6 Evaluation

We evaluate TIERSizer to answer two questions: (i) how much DRAM does it save while keeping performance within the target? and (ii) how accurate are its counterfactual models?

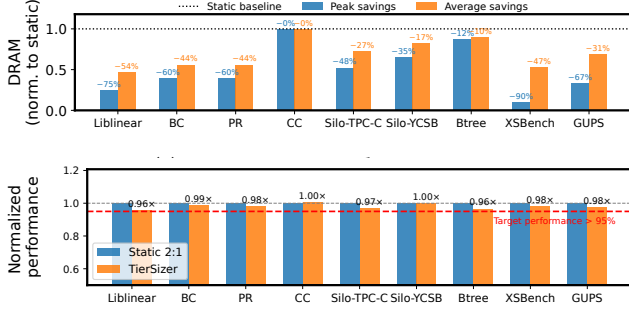
Hardware Setup. We run our experiments on a c220g5 node in (Cloudlab [10]) running Linux v6.18. We use remote NUMA memory to emulate the slow CXL tier. We lower the uncore frequency of the remote node to emulate longer slow tier latencies. Unloaded latency is ≈ 90 ns for local DRAM and ≈ 180 ns for the emulated CXL tier.

Workloads. For our evaluation, we select a set of eight representative and diverse workloads shown in Table 1. These workloads have been widely used to evaluate tiered memory systems in many prior works [18, 19, 29, 37]. We run these workloads with 16 threads, large enough to saturate the memory bandwidth of each system.

6.1 DRAM savings

We compare TIERSizer against the static 2:1 DRAM:CXL ratio operators recommend today, and set the performance target to a 5% slowdown relative to this 2:1 baseline.

Figure 3 shows our main result. TIERSizer reduces DRAM by 34.6% on average relative to the 2:1 baseline while keeping every workload within the 5% slowdown target. TIERSizer is able to eliminate the slack in DRAM allocations that exist in systems today. We observe that TIERSizer achieves different savings with different workloads. The amount saved depends mainly on the application's sensitivity to DRAM. For example, XSbench, whose hot set is tiny, sees the largest reduction, upto 90%. TIERSizer predicts its slowdown across sizes, keeps shrinking DRAM, and stops just before the cliff shown in Figure 1.



(b) Performance with TIERSizer.

Figure 3. DRAM savings and performance achieved with TIERSizer. All workloads are within the target slowdown of $<5\%$ from the static size baseline.

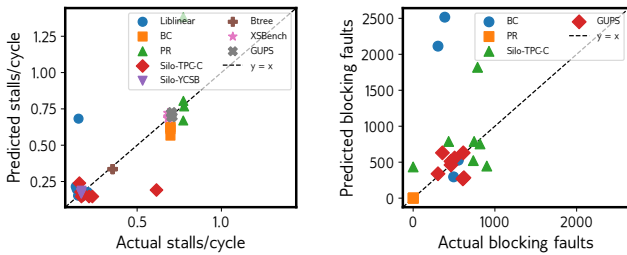


Figure 4. Accuracy of TIERSizer's two models.

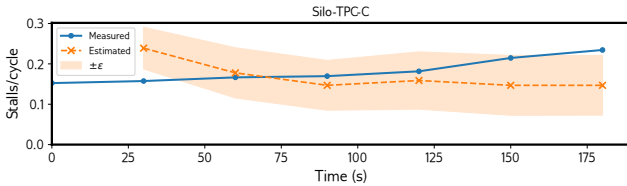


Figure 5. Comparison of actual stalls versus estimated stalls for Silo-TPC-C.

TIERSizer is able to save memory for all workloads except GapBS-CC. TIERSizer cannot fit an accurate model for CC; the residual error ϵ remains high ($\approx 20\%$), so TIERSizer is unable to resize and CC simply retains its baseline allocation. We attribute this to CC's stalls varying continuously, faster than TIERSizer's control period can track. Notably, the failure is *safe*: an unreliable model yields no resize rather than a harmful one, so CC stays within target at the cost of forgone savings.

6.2 Model prediction accuracy

The savings achieved are entirely dependent on TIERSizer predicting slowdown accurately at sizes the workload is *not* running on. Figure 4 compares predicted and measured values for both models (stalls and blocking faults) across workloads and DRAM sizes. Because the prediction is a counterfactual, we evaluate it retrospectively. At the end of each period, TIERSizer predicts stalls for the next period at DRAM size D and the neighboring sizes $D - \Delta D$ and $D + \Delta D$. The controller then decides to stay at D or move to a neighboring size. Once that period completes, we compare the measured stalls against

the prediction made for the size the controller selected. Note that predictions for sizes not selected cannot be validated this way, since the workload never runs there; what we validate is precisely the prediction the controller acts on.

Stalls model. The median error in the stall estimate is 2.27% (not shown in figure). More than the magnitude, what matters is the *direction* of error. Over-predicting slowdown costs a little DRAM, whereas under-predicting risks violating the target. Most under-predictions are small and are within the epsilon error margin, with one notable exception in Silo-TPC-C.

We examine Silo-TPC-C closely in Figure 5, which plots the estimated stalls against those measured over time. Although TIERSizer under-predicts, the measured stalls stay within the $\pm\epsilon$ band most of the time. The residual margin absorbs exactly the kind of error a single point estimate would miss. The band, however, *widens* over time, which is informative that the model is no longer a good fit. Silo-TPC-C is write-heavy and its RSS grows over time. Moreover, it follows a recency-skewed (latest) distribution where newly inserted entries are hot, representing a continuously changing hot set. This reveals a main limitation of TIERSizer. TIERSizer cannot estimate correctly for workloads with continuously changing hot sets and growing footprints, where no recent window predicts the next.

Migration-overhead model. The overhead model's accuracy depends mainly on estimating the number of blocking faults. As Figure 4 shows, the estimated counts track the measured counts closely, and the larger deviations (e.g., for BC) are over-predictions, which keep the slowdown estimate conservative rather than risking a violation. Because migration overhead is a small fraction of total slowdown, these errors have limited effect on the final prediction.

7 Conclusion

Memory efficiency matters more than ever in datacenters. With tiered systems, the key question is how much DRAM to allocate to an application. We address this by answering a counterfactual—what would an application's performance be at different DRAM sizes. We build two simple models that estimate the main contributors to slowdown: LLC-miss stalls and migration overhead, while accounting for imperfect tiering. We design a controller that uses these models to predict slowdown at counterfactual sizes and shrink DRAM as far as it can while keeping performance within a specified target.

Acknowledgments

This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant Number 2326576, and NSF grants 2403026 and 2422130; also supported by the InfraAI Center at The University of Texas at Austin. Any opinions, findings and conclusions or recommendations expressed in this material do not necessarily reflect the views of the U.S. National Science Foundation.

References

- [1] Reto Achermann and Ashish Panwar. 2020. Mitosis Workload Btree. <https://github.com/mitosis-project/mitosis-workload-btree>
- [2] Neha Agarwal and Thomas F Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [3] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).
- [4] Shangye Chen, Jin Huang, Shuangyan Yang, Jie Liu, Huaicheng Li, Dimitrios Nikolopoulos, Junhee Ryu, Jinho Baek, Kwangsik Shin, and Dong Li. 2024. Tuning Fast Memory Size based on Modeling of Page Migration for Tiered Memory. *arXiv preprint arXiv:2410.00328* (2024).
- [5] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. 2015. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 213–224.
- [6] Compute Express Link Consortium. Inc. 2024. CXL 3.2 Specification. (2024). <https://computeexpresslink.org/cxl-specification/>
- [7] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. *LWN* (2012). <https://lwn.net/Articles/488709/>
- [8] Thaleia Dimitra Doudali and Ada Gavrilovska. 2022. Coeus: Clustering (a) like patterns for practical machine intelligent hybrid memory management. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 615–624.
- [9] Thaleia Dimitra Doudali, Daniel Zahka, and Ada Gavrilovska. 2021. Cori: Dancing to the right beat of periodic data movements over hybrid memory systems. In *2021 IEEE International Parallel and Distributed Processing Symposium*.
- [10] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *2019 USENIX Annual Technical Conference*.
- [11] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. 2023. 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*.
- [12] Pouya Esmaili-Dokht, Francesco Sgherzi, Valéria Soldara Girelli, Isaac Boixaderas, Mariana Carmin, Alireza Monemi, Adria Armejach, Estanislao Mercadal, Germán Llorca, Petar Radojković, et al. 2024. A mess of memory system benchmarking, simulation and application profiling. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.
- [13] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *The Journal of machine Learning research* (2008).
- [14] Neha Gholkar, Jovan Stojkovic, Hasan Al Maruf, Gregory Price, Prakash Chauhan, Hiral Patel, Cedric Van Goethem Kiran Vemuri, Kiran Malwankar, Kishore Sriadibhatla, Kalyan Subramanian, et al. 2026. Vistara: Making CXL Real-Full Path From ASIC Design and OS Support to Hyperscale Deployment. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE.
- [15] 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*.
- [16] Intel Corporation. 2018. Intel 64 and IA-32 Architectures Software Developer Manuals. (2018). <https://software.intel.com/articles/intel-sdm>
- [17] Konstantinos Kanellis, Sujay Yadalam, Shivaram Venkataraman, and Michael Swift. 2025. Striking the Right Chord: Parameter Tuning in Memory Tiering Systems. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems*.
- [18] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*.
- [19] Baptiste Lepers and Willy Zwaenepoel. 2023. Johnny Cache: the End of {DRAM} Cache Conflicts (in Tiered Main Memory Systems). In *17th USENIX Symposium on Operating Systems Design and Implementation*.
- [20] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.
- [21] Jinshu Liu, Hamid Hadian, Yuyue Wang, Daniel S Berger, Marie Nguyen, Xun Jian, Sam H Noh, and Huaicheng Li. 2025. Systematic cxl memory characterization and performance analysis at scale. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [22] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered memory management beyond hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.
- [23] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent page placement for CXL-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [24] Timothy Morgan. 2020. CXL And Gen-Z Iron Out A Coherent Interconnect Strategy. (2020). <https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/>
- [25] Seongjae Park, Yunjae Lee, and Heon Y Yeom. 2019. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*.
- [26] Steven J Plimpton, Ron Brightwell, Courtenay Vaughan, Keith Underwood, and Mike Davis. 2006. A simple synchronous distributed-memory algorithm for the HPCC RandomAccess benchmark. In *2006 IEEE International Conference on Cluster Computing*.
- [27] PuLP. [n. d.]. PuLP: A Linear Programming Toolkit for Python. <https://coin-or.github.io/pulp/>. Accessed: 2025-07-08.
- [28] Milan Radulovic, Rommel Sánchez Verdejo, Paul Carpenter, Petar Radojković, Bruce Jacob, and Eduard Ayguadé. 2019. PROFET: Modeling system performance and energy without simulating the CPU. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2019).
- [29] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*.
- [30] Kevin Song, Jiacheng Yang, Zixuan Wang, Jishen Zhao, Sihang Liu, and Gennady Pekhimenko. 2025. HybridTier: an Adaptive and Lightweight CXL-Memory Tiering System. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [31] Wenda Tang, Yiduo Wang, Yanwen Wang, and Jie Wu. 2025. Leave No One Behind: Fair and Efficient Tiered Memory Management for Multi-Applications. In *Proceedings of the 54th International Conference on Parallel Processing*. 699–709.
- [32] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future* (2014).
- [33] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the ACM*

SIGOPS 30th Symposium on Operating Systems Principles.

- [34] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. 2022. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [35] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. {FlexMem}: Adaptive page profiling and migration for tiered memory. In *2024 USENIX Annual Technical Conference*.
- [36] Sujay Yadalam, Konstantinos Kanellis, Michael Swift, and Shivaram Venkataraman. 2025. ARMS: Adaptive and Robust Memory Tiering System. *arXiv preprint arXiv:2508.04417* (2025).
- [37] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. 2017. AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter. In *2017 IEEE 36th International Performance Computing and Communications Conference*.
- [38] Xinyue Yi, Hongchao Du, Yu Wang, Jie Zhang, Qiao Li, and Chun Jason Xue. 2025. ArtMem: Adaptive Migration in Reinforcement Learning-Enabled Tiered Memory. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*.
- [39] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, et al. 2024. Managing memory tiers with {CXL} in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*.