Tiered Memory Management: Access Latency is the Key!

Midhul Vuppalapati
Cornell University

Rachit Agarwal
Cornell University

Abstract

The emergence of tiered memory architectures has led to a renewed interest in memory management. Recent works on tiered memory management innovate on mechanisms for access tracking, page migration, and dynamic page size determination; however, they all use the same page placement algorithm—packing the hottest pages in the default tier (one with the lowest hardware-specified memory access latency). This makes an implicit assumption that, despite serving the hottest pages, the access latency of the default tier is less than that of alternate tiers. This assumption is far from real: it is well-known in the computer architecture community that, in the realistic case of multiple in-flight requests, memory access latency can be significantly larger than the hardware-specified latency. We show that, even under moderate loads, the default tier access latency can inflate to be 2.5× larger than the latency of alternate tiers; and that, under this regime, performance of state-of-the-art memory tiering systems can be 2.3× worse than the optimal.

Colloid is a memory management mechanism that embodies the principle of balancing access latencies—page placement across tiers should be performed so as to balance their average (loaded) access latencies. To realize this principle, Colloid innovates on both per-tier memory access latency measurement mechanisms, and page placement algorithms that decide the set of pages to place in each tier. We integrate Colloid with three state-of-the-art memory tiering systems—HeMem, TPP and MEMTIS. Evaluation across a wide variety of workloads demonstrates that Colloid consistently enables the underlying system to achieve near-optimal performance.

CCS Concepts: • **Software and its engineering** → **Memory management**; • **Computer systems organization** → **Heterogeneous (hybrid) systems**.

Keywords: Operating Systems, Tiered Memory Management

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695968

ACM Reference Format:
Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3694715.3695968

1 Introduction

Modern memory-intensive applications such as in-memory databases, graph processing engines, and machine learning frameworks benefit from larger memory capacity and memory interconnect bandwidth. Indeed, memory contributes to increasingly larger fractions of cloud server costs (*e.g.*, ~37% for Meta [35] and ~50% for Microsoft Azure [38]). Unfortunately, technology trends for the classical memory architecture—DRAM modules exposed to the processor via DDR memory interconnect—have become stagnant, making it hard to scale both memory capacity and memory interconnect bandwidth in a cost-effective manner [41, 46, 49, 54, 59]. The situation for memory interconnect bandwidth is particularly dire: as number of cores and concurrency-per-core continue to increase, the memory interconnect is becoming increasingly contended [2, 36]. For instance, across the most recent generations of Intel servers—Intel Cascade Lake (2019), Ice Lake (2021) and Sapphire Rapids (2023)—processors are able to generate 2.98×, 3.31× and 4.5× larger traffic than the memory interconnect bandwidth, respectively; for AMD Genoa (2023), processors can generate 9.37× larger traffic than the memory interconnect bandwidth.

Classical memory architecture reaching its scaling limits has led to the emergence of tiered memory architectures. These architectures expose to the processor, in addition to the classical DDR-attached memory, new forms of cache-coherent memory via alternate memory interconnects [10, 32, 35, 54, 62]. For instance, Compute Express Link (CXL) enables processors to access cache-coherent memory over the serial interface technology of peripheral interconnects (*e.g.*, PCIe), thus enabling a new memory tier with additional memory bandwidth but with higher latency [10, 54, 62]. Emergence of such tiered memory architectures has led to renewed interest in memory management since page placement across memory tiers critically impacts application performance.

Recent work on memory management for tiered memory architectures [1, 12, 24, 29, 35, 48, 60, 61] has led to many innovative mechanisms for access tracking, page migration, dynamic page size determination, etc.; however, they all use the same page placement algorithm—packing as many hot pages as possible in the memory tier with the lowest unloaded access

latency[1] (referred to as the default tier), with the remaining pages placed in alternate tiers. This page placement algorithm is based on a core implicit assumption that, despite serving the hottest pages, memory access latency of the default tier is less than that of alternate tiers. This assumption is far from real: it is well-known in the computer architecture community that, in the realistic case of multiple in-flight requests, memory access latency can be significantly larger than the unloaded latency, even when the memory interconnect bandwidth is far from saturated [13, 42–44, 58]; see §3. We refer to this regime as memory interconnect contention. Indeed, we demonstrate that memory interconnect contention can lead to 5× inflation in access latency of the default tier even under moderate loads. Given the access latency of existing CXL hardware, such inflation would result in the default tier having 2.5× higher latency than alternate tiers. We show in §2 that, under memory interconnect contention, performance of state-of-the-art memory tiering systems can be 2.3× worse than the optimal.

We present Colloid, a tiered memory management mechanism that embodies the principle of balancing access latencies—page placement across tiers should be performed so as to balance their average (loaded) access latencies. The principle of balancing access latencies builds upon a simple observation: placing more hot pages in the tier with lower access latency may increase the access latency of the tier and decrease the access latency of other tiers (thus, more closely balancing the access latencies of the tiers); however, it will reduce the overall average memory access latency. This observation is important because each processor core is able to keep a bounded number of memory requests in-flight; thus, minimizing average memory access latency directly maximizes memory access throughput and application-level performance (§3).

The principle of balancing access latencies provides a unified approach to memory management. First, it naturally captures the *unloaded* latency of the tiers—if the (loaded) access latency of the default tier is smaller than that of alternate tiers, then balancing access latencies requires increasing the fraction of accesses to the default tier, thus converging to the page placement algorithm in existing systems. Second, it captures the fact that memory interconnect contention can happen even when memory bandwidth is far from saturated—memory interconnect contention at any point within the CPU-to-memory datapath will result in access latency increasing for the corresponding tier, and will thus automatically get factored in while making page placement decisions based on the principle.

Memory management based on the principle of balancing access latency changes the core structure of the tiered memory management problem. Indeed, it is no longer optimal to pack as many hot pages as possible in the default tier since placing hot pages in the alternate tier may result in improved application performance. We thus need new mechanisms to measure memory access latency, and new page placement algorithms that decide the set of pages to place in each tier. Colloid innovates along both these directions. First, Colloid demonstrates that modern servers have vantage points within the CPU-to-memory datapath that can be leveraged to perform low-overhead measurements of per-tier queue occupancy and request arrival rates; using Little's Law, this enables Colloid to measure per-tier memory access latency at fine-grained timescales. Second, Colloid presents a new page placement algorithm that takes per-tier memory access latency and per-page access tracking information as input, and uses the principle of balancing access latencies to efficiently decide the set of (hot and cold) pages to place at each tier.

Colloid design is compatible with any cache-coherent tiered memory architecture where different tiers do not share memory channels; this includes local DDR-attached memory, remote socket memory exposed through the processor interconnect, CXL-attached memory, and High Bandwidth Memory [19, 37]. Colloid design is also compatible with existing mechanisms for access tracking, page migration and page size determination, thus enabling Colloid to easily integrate with existing memory tiering systems. We demonstrate this by integrating Colloid with state-of-the-art memory tiering systems—HeMem [48], TPP [35], and MEMTIS [29]. We evaluate these implementations using real-world applications and using standard workload generators over a range of static and time-varying workloads with varying memory interconnect contention, varying core counts, varying object sizes, varying memory access latencies, and varying read/write characteristics. Across all evaluated scenarios, we find that Colloid enables each system to achieve near-optimal performance, independent of the memory interconnect contention intensity; we also find that Colloid does not impact the convergence time for the underlying system under time-varying workloads.

Colloid implementation for each system, along with the documentation to reproduce all our results, is available at `https://github.com/host-architecture/colloid`.

2 Motivation

In this section, we study three state-of-the-art memory tiering systems—HeMem [48], TPP[2] [35] and MEMTIS [29]—and demonstrate that:

- Under the realistic case of multiple in-flight memory requests, access latency of the default tier can be 5× higher than its unloaded latency. We refer to this regime as the memory interconnect contention regime (defined precisely in §3.1). In our setup, the default tier latency in the memory interconnect contention regime can inflate to be 2.4× higher than the alternate tier latency. Based on latencies reported for real CXL hardware [54, 62], a 5×

[1] The unloaded access latency is defined as the memory access latency when there is only one request in-flight. In contrast, the loaded access latency is defined as the access latency when there are multiple requests in-flight.

[2] For TPP, we use Linux kernel v6.3 that includes TPP upstreamed version along with several improvements [16, 27]. We enable Transparent Huge Pages.

inflation in default tier access latency would correspond to 2.5× higher latency than the alternate tier.

- Existing memory tiering systems pack as many hot pages as possible in the default tier. Under memory interconnect contention regime, this turns out to be a suboptimal strategy, resulting in these systems performing far from optimal—as much as 2.3×, 2.36× and 2.46× worse performance than optimal for HeMem, TPP and MEMTIS, respectively.

2.1 Experimental Setup

We use a dual-socket server, with each socket having an Intel Xeon Platinum 8362 CPU with 32 cores, 1.25MB L2 cache per-core, 48MB LLC, and 8× 3200MHz DDR4 memory channels with one DIMM attached per channel. Thus, the total theoretical maximum bandwidth across all channels is 205GB/s. Sockets are connected by a UPI link with 75GB/s theoretical maximum bandwidth (in each direction). We use processors only on one of the sockets; the default tier is the locally-attached memory of the socket (32GB capacity, and 70ns unloaded latency), and the alternate tier is the memory attached to the other socket (96GB capacity, and 135ns unloaded latency).

We use the GUPS workload from [48] adapted to our setup. The working set consists of a virtually contiguous buffer of size 72GB. A random 24GB region of this buffer constitutes the hot set; thus, the hot set fits in the default tier but the full working set does not. Cores $1-15$ run one thread each, reading and updating ($1:1$ RW ratio) a 64 byte object chosen at random from the hot set with 90% probability and from the full working set with 10% probability. Cores $31-32$ are used for samping and migration threads used in memory tiering systems. We do not oversubscribe CPU cores as our evaluation does not focus on evaluating CPU overheads of existing systems; that has already been studied in prior works [29, 48].

We focus on memory interconnect contention on the default tier; memory interconnect contention on the alternate tier does not break the assumption of default tier latency being lower than the alternate tier latency—existing systems already perform ideal page placement under such a regime. To generate controlled memory interconnect contention, as in prior works [2, 3, 8, 58], we use a memory antagonist on cores $16-30$ that generates sequential $1:1$ read/write memory traffic to a 500MB buffer that is pinned to the default tier memory. To study behavior of the systems in steady-state and provide deeper insights into the observed results, we generate constant memory interconnect contention throughout the experiment; we study the behavior of systems under dynamic changes in memory interconnect contention later (§5). Across experiments, we vary the intensity of the memory interconnect contention by varying the number of cores used to run the memory antagonist—0×,1×,2×,3× intensities correspond to $0, 5, 10, 15$ cores, with memory bandwidth usage of 0%, 51%, 65%, 70%, respectively, when run in isolation. We allow enough time so that each system reaches steady-state, and measure steady-state application throughput.

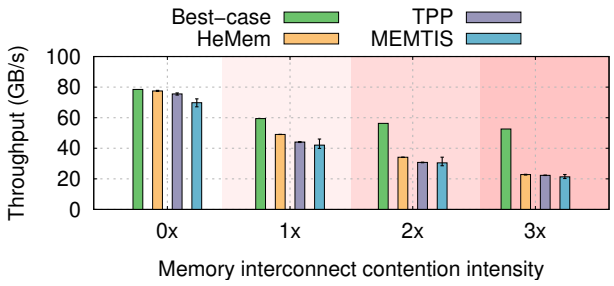

Figure 1. Even at moderate memory interconnect contention intensity, existing memory tiering systems achieve performance that is far from optimal. Each bar shows GUPS throughput averaged across 3 runs, with error bars showing the minimum and maximum values across the runs.

We determine the best-case memory placement for each configuration by manually placing $0-100$% of the hot set in the default tier (in increments of 10) using the Linux `mbind` API; the remaining hot set is placed in the alternate tier and any remaining capacity in the default tier is filled with randomly chosen pages from the cold set. We call the highest throughput across these manual placements as the best-case application throughput.

2.2 Understanding impact of memory interconnect contention on existing memory tiering systems

Figure 1 shows the steady-state throughput for each system alongside the best-case throughput with varying intensity of memory interconnect contention. With 0× memory interconnect contention intensity, HeMem, TPP, and MEMTIS achieve throughput within 1.5%, 4.6% and 10.1% of the best-case respectively. MEMTIS automatically decides whether to use 4KB or 2MB pages for different parts of the working set based on access tracking information. Here, it incurs additional performance degradation because it splits 2MB pages into 4KB pages even though it is not beneficial for this workload. This is because it ends up making hugepage splitting decisions before the workload has reached steady-state and is unable to coalesce pages that have been split; digging deeper, we found that MEMTIS performs page coalescing using an inefficient mechanism of scanning the virtual address space which takes significantly longer than the time it takes for this workload to reach steady-state.

With increasing memory interconnect contention, the throughput achieved by these systems begins to diverge from the best-case. Even with 1× memory interconnect contention intensity, HeMem, TPP and MEMTIS throughput is 1.21×, 1.35×, and 1.41× lower than the best-case, respectively. The throughput gap increases with higher memory interconnect contention intensities, reaching as high as 2.3×, 2.36×, 2.46× for HeMem, TPP and MEMTIS, respectively.

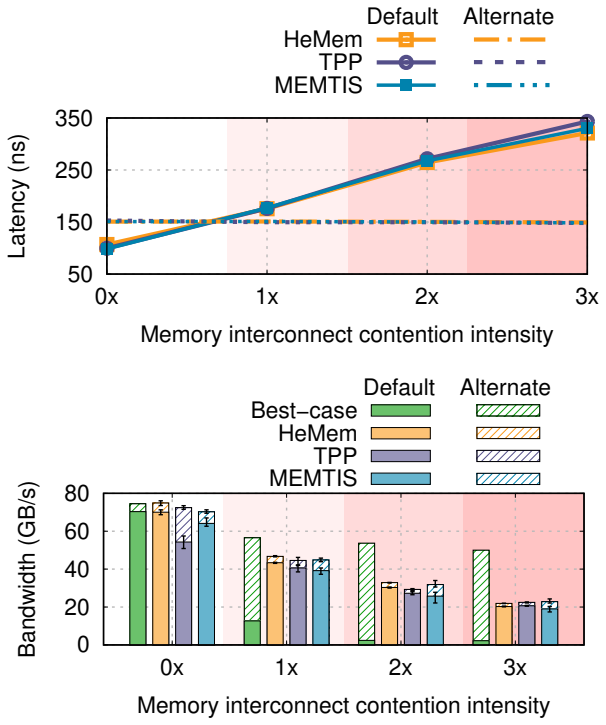

Figure 2. Understanding the root cause of suboptimal performance of existing memory tiering systems in Figure 1 (a-b; top-bottom). (a) Even with moderate memory interconnect contention intensity, the average memory access latency of the default tier exceeds that of the alternate tier. (b) Existing memory tiering systems place nearly the entire hot set in the default tier independent of memory interconnect contention intensity, while the optimal placement entails placing an increasingly larger fraction of the hot set in the alternate tier with increasing memory interconnect contention.

Under memory interconnect contention, default tier access latency can exceed that of alternate tier. We measure the average memory access latency for each tier during the above experiments using hardware counters in the processor Caching and Home Agent (CHA); see §3.1 for details. Figure 2(a) demonstrates that the assumption made in all prior work—despite serving the hottest pages, default tier having lower memory access latency than the alternate tier—does not hold. Indeed, we find that, with memory interconnect contention increasing from 1× to 2× to 3×, the default tier's access latency increases by 2.5×, 3.8× and 5×, respectively. In our setup, this corresponds to default tier access latency exceeding that of the alternate tier by 1.2×, 1.8× and 2.4×, respectively. Access latency increases due to queueing of requests at the memory controller which can happen even when memory bandwidth is far from saturated (see §3.1 for more detailed discussion).

Existing systems continue to greedily place hottest pages in default tier under memory interconnect contention. Figure 2(b) shows the memory bandwidth usage of GUPS across individual tiers for the best-case and for each system, measured using Intel's Memory Bandwidth Monitoring (MBM) feature. For the best-case, the default tier bandwidth accounts for only 25%, 4.5%, 4% of total bandwidth at 1×, 2× and 3× intensities of memory interconnect contention respectively—corresponding to increasingly larger fractions of the hot set being placed in the alternate tier. Intuitively, when the default tier access latency exceeds that of the alternate tier, it is no longer optimal to place the entire hot set in the default tier. However, for HeMem, TPP and MEMTIS, default tier bandwidth accounts for more than 90%, 75% and 85% of total bandwidth independent of memory interconnect contention intensity—indicating that they always place a majority of the hot set in the default tier. Such memory interconnect contention agnostic page placement is the root cause for their throughput becoming increasingly far from optimal. Specifically, since each core can maintain a fixed number of in-flight memory requests [55, 58], per-core throughput (T) is given by $\frac{N \cdot 64}{L}$ where N is the number of in-flight requests, L is the access latency and 64 is the size of each memory request in bytes (that is, 1 cacheline). From 0× to 3× memory interconnect contention intensity, default tier access latency increases by ~3.5× and throughput of each of the systems reduces by a similar factor (3.42×, 3.39× and 3.29× for HeMem, TPP and MEMTIS respectively).

3 Colloid

We now present the Colloid design. We start with the core underlying principle in the Colloid design—the principle of balancing access latencies—and discuss how it provides a unified approach to guide page placement in tiered memory architectures (§3.1). We then describe how Colloid realizes this principle into an end-to-end tiered memory management mechanism. Specifically, we describe an efficient mechanism to measure per-tier memory access latency at fine-grained timescales (§3.1) and a new algorithm that performs dynamic page placement across tiers based on access latencies (§3.2).

We consider a tiered memory architecture illustrated in Figure 3. Memory in all tiers is exposed in the host physical address space, and can be accessed by the CPU cores through load/store instructions in a cache-coherent manner with the same memory consistency model. The tier with the smallest unloaded latency is called the default tier, and the others are collectively called alternate tiers. Access to each tier is facilitated through a separate memory controller. All the above properties hold for existing tiered memory architectures where the default tier is local DDR-attached memory, and the alternate tiers are either remote socket memory exposed through the processor interconnect, CXL-attached memory, and/or High Bandwidth Memory.

To keep the discussion succinct, we assume that all mechanisms within the memory tiering system (access tracking, latency measurements, page placement, etc.) are performed periodically at fixed time intervals, referred to as quantums. As we discuss in §4, Colloid can be easily

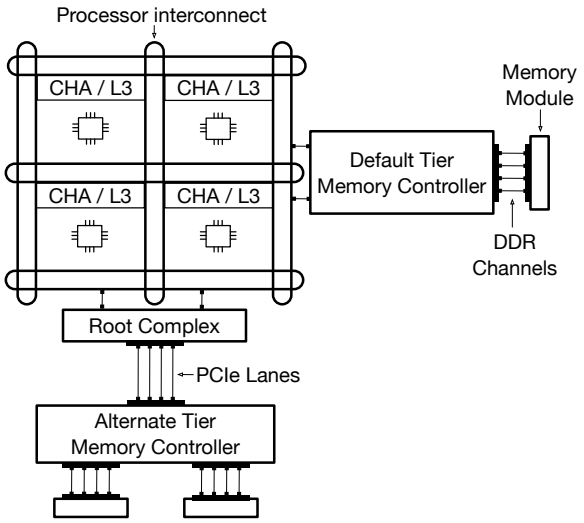

Figure 3. Illustration of a tiered memory architecture with two memory tiers. Cores, L3 cache, Caching and Home Agent (CHA), and default tier memory controller are connected through the on-chip processor interconnect. Both the L3 cache and CHA are physically distributed into multiple slices (co-located with cores). The physical path between the processor and alternate tier memory controller depends on the type of alternate memory tier. The figure shows CXL-attached memory as an example. Here, the first hop is the root complex on the processor interconnect which is connected to an external memory controller through PCIe lanes.

integrated with systems that use various triggers (*e.g.*, page faults) for access tracking and/or page placement.

3.1 Access latency is the key

The key conceptual idea underlying the Colloid design is the *principle of balancing access latencies*. The principle suggests that page placement across tiers should be performed so as to balance their average access latencies.

A memory request refers to a memory access issued by a core that misses all levels of the cache hierarchy and is serviced from either the default or the alternate tier. We define the unloaded latency as the latency when there is only one in-flight memory request in the system; and, loaded latency as the latency under multiple concurrent memory requests. It is well-known that the loaded latency for both default and alternate tiers can increase beyond their unloaded latency due to contention within the CPU-to-memory datapath:

- Latency can increase due to queueing of requests when the corresponding interconnect bandwidth is saturated [13, 14, 21, 25, 26, 39, 40, 42–45, 51–54]. Memory bandwidth utilization at the saturation point is hard to characterize in advance since it can vary by as much as 1.75× depending on whether the workload is read-heavy or write-heavy, and be as much as 2.5× lower than the theoretical maximum bandwidth [54].
- Latency can also increase even when the interconnect bandwidth is far from saturated [13, 25, 26, 39, 40, 42–45, 51–53, 58]. This happens due to contention within the internal hierarchy of memory modules. For example, each DRAM module consists of multiple banks; load imbalance across these banks and lack of locality in access patterns within each bank can result in queueing of requests at the memory controller, leading to latency inflation [13, 40, 43–45, 58].

We collectively refer to the regime of memory access latency increasing beyond the unloaded latency as memory interconnect contention. We define the access probability of a page during a given quantum as the total number of memory requests to that page normalized by the total number of memory requests across all pages during that quantum. During a given quantum, let L_D, L_A be the average access latencies of the default and alternate tiers, respectively, and let p be the sum of access probabilities of pages currently in the default tier.

The principle of balancing access latencies suggests that:

- If $L_D < L_A$, page placement should be adapted to increase p (*e.g.*, by placing more hot pages in the default tier).
- If $L_D > L_A$, page placement should be adapted to reduce p (*e.g.*, by placing more hot pages in the alternate tier).
- If $L_D = L_A$, page placement does not need to be adapted.

The reasoning behind the principle of balancing access latencies is simple. The performance of memory-intensive applications is bound by the overall memory access throughput (rate at which memory requests are serviced). The maximum number of in-flight memory requests that each core can maintain (N) is limited by hardware buffer sizes (Line Fill Buffers [55, 58]). As a result, the average per-core memory access throughput (T) is directly related to the average memory access latency (L) by $T = \frac{N \cdot 64}{L}$, where 64 is the size of each memory request in bytes (that is, one cacheline). Therefore, minimizing average access latency maximizes throughput. The average memory access latency is given by $p \cdot L_D + (1-p) \cdot L_A$. If $L_D < L_A$, then increasing p reduces the average access latency; on the other hand, if $L_D > L_A$, then decreasing p reduces the average access latency. Finally, if $L_D = L_A$, then changing p does not change the average access latency. Overall, $L_D = L_A$ is indeed the equilibrium point to minimize average access latency and maximize throughput.

The principle of balancing access latencies provides a unified approach to guide page placement across tiers. First, it automatically captures unloaded latencies of each tier—these are included in L_D and L_A, respectively. For example, if the gap between unloaded latencies of the tiers is larger, then L_D will remain less than L_A until larger degree of contention on the default tier and the principle of balancing access latencies will suggest placing a larger amount of hot pages in the default tier. Second, it captures the maximum theoretical bandwidth of each tier. Higher tier bandwidth usually implies that higher loads can be handled while keeping access latency closer to the unloaded latency (and vice versa). If

the bandwidth of any tier is saturated, then its corresponding access latency will increase, and thus get captured by the principle of balancing access latencies. Third, it captures the impact of memory interconnect contention even when bandwidth is not saturated—queueing at any point in the memory access datapath will result in access latency increase for the corresponding tier, and will thus get factored into page placement decisions based on the principle.

The principle of balancing access latencies naturally generalizes to tiered memory architectures with more than two tiers. If the access latencies of all the tiers are not equal, then the average access latency can be reduced by placing more hot pages in the tier with the smallest access latency. This is because doing so will increase the sum of access probabilities of pages in this tier while correspondingly reducing the sum of access probabilities of pages in other tier(s) which have higher access latencies. Similar reasoning can be applied recursively for the tier with the second smallest access latency and so on. If the access latencies of all the tiers are equal, then adapting page placement will not lead to change in average access latency. Hence, the state of balanced access latency across tiers remains to be the equilibrium point that minimizes average access latency and maximizes throughput.

Measuring access latency. Recent generations of Intel and AMD processors provide hardware counters that enable low-overhead measurements of per-tier access latency [4, 17, 18, 20]. For brevity, we focus our discussion on Intel processors.

Independent of the read/write ratio of the workload, overall memory access throughput primarily depends on the latency of memory read requests. This is because, even for write requests, store instructions first generate memory read upon cache miss to load data into the cache; the data is then updated in the cache and memory write is processed asynchronously upon cache writeback [28, 58]. Thus, the memory access throughput for write requests directly depends on the latency of memory read requests. To that end, we just need mechanisms to measure the latency of memory read requests.

Colloid leverages the processor's Caching and Home Agent (CHA) as a vantage point to measure the access latency of each tier. The CHA abstracts away memory tiers from the rest of the system while handling cache-coherence. As shown in Figure 3, the CHA is physically partitioned into multiple slices (colocated with cores), each of which owns a disjoint subset of the host physical address space. Upon an L1/L2 cache miss, memory requests are first forwarded to their corresponding CHA slice based on their physical address. The CHA looks up the L3 cache. Upon a miss, the CHA queues the request in its buffers and then forwards the request to the default or alternate tier based on the physical address. The request remains queued at the CHA until it has been serviced from the corresponding tier.

CHA hardware counters enable low-overhead measurements of queue occupancy and request arrival rates on a per-request type (read/write) and per-tier basis for local DDR-attached memory, remote socket memory exposed through the processor interconnect, CXL-attached memory, as well as High Bandwidth Memory. These measurements can be performed at much finer-grained timescales (as low as 1 microsecond) than the timescales at which memory tiering systems typically make page placement decisions. During a given time quantum, let O_D, O_A be the average queue occupancy of default and alternate tier requests, respectively, and let R_D, R_A be the average arrival rate of requests to the default and alternate tiers (number of memory requests that arrived during the quantum divided by the quantum duration). R_D and R_A are directly related to p (sum of access probabilities of pages in the default tier): $p = \frac{R_D}{R_D+R_A}$. Colloid uses Little's Law to measure the access latency (L_D, L_A) of each tier: $L_D = O_D/R_D$, $L_A = O_A/R_A$. Since Little's Law applies to any system where queues do not grow unboundedly (without any assumptions on arrival, service behaviors, scheduling policies etc.), applying it at the CHA gives us the average CHA to memory latency; recent work [58] provides in-depth validation of Little's law based memory access latency measurements. The only missing component in measured L_D, L_A is CPU to CHA latency. This accounts for only a tiny fraction of the CPU to memory access latency and can thus be ignored. For example, on our hardware setup, out of ~70ns unloaded latency for default tier, ~5ns is CPU to CHA and ~65ns is CHA to DRAM; under memory interconnect contention, the latter will become even higher, making the former an even smaller fraction of the overall latency. We apply Exponentially Weighted Moving Averaging (EWMA) on the both the occupancy and rate measurements to smooth noise in the signals. This trades-off slightly higher reaction time during workload changes for better stability.

3.2 Colloid page placement algorithm

Colloid page placement algorithm takes per-tier access latency and per-page access tracking information as input, and adapts page placement across tiers.

Overview of the Colloid page placement algorithm. Algorithm 1 shows the end-to-end Colloid page placement algorithm. At the beginning of each quantum, it obtains the per-tier average queue occupancy (O_D, O_A) and average request rates (R_D, R_A) over the duration of the previous time quantum, computes the per-tier access latencies (L_D, L_A), and current value of the sum of access probabilities of pages in the default tier p. It then decides where to migrate pages based on the principle of balancing access latencies (§3.1)—if default tier access latency is smaller than that of the alternate tier, then it promotes hot pages from the alternate tier to the default tier; otherwise it demotes hot pages from the default tier to the alternate tier (Lines 5-8). Next, it determines what pages to migrate in three steps. First, it computes how much access probability to shift between the

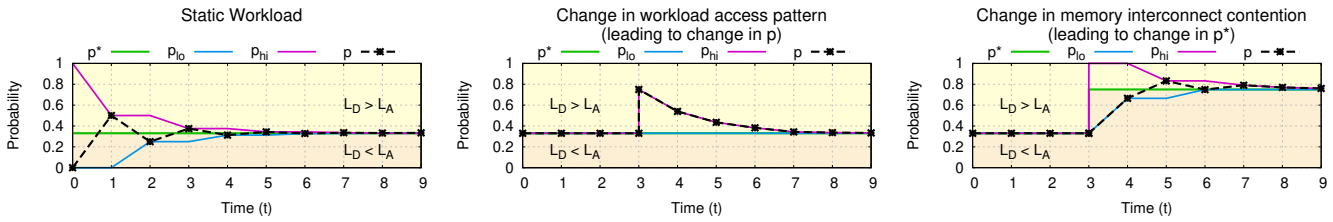

Figure 4. Conceptual illustration of Colloid page placement algorithm (a-c; left-right). Each figure shows the change in sum of access probabilities of pages in default tier p over time, along with the equilibrium point p^*, low watermark p_{lo}, and high watermark p_{hi}. Colloid page placement algorithm adapts p, p_{lo} and p_{hi} so as to maintain two invariants: both p and p^* are contained between p_{lo} and p_{hi} at all times, and the gap between p_{lo} and p_{hi} reduces over time. (a) Static workload: p eventually converges to $p*$ (b) Upon sudden change in p at $t=3$, p_{hi} is *updated*, after which p eventually converges to $p*$ (c) Upon sudden change in $p*$ at $t=3$, p_{hi} is *reset*, after which p eventually converges to $p*$.

tiers, denoted as Δp (Line 9). Second, it computes a migration limit. Third, it executes a page finding procedure (Line 10), whose goal is to find a set of pages in default/alternate tier for demotion/promotion under the following two constraints: (1) their sum of access probabilities is less than or equal to Δp; and (2) their sum of sizes in bytes does not exceed the migration limit or the available capacity in the destination tier. Depending on the underlying system and access tracking mechanism, one can implement different page finding procedures, which we describe in §4. Finally, it migrates these pages through an underlying page migration mechanism.

Algorithm 1 : Colloid page placement algorithm.

[Input parameter] M: Migration limit per-quantum (in bytes)
Every quantum:

1: O_D, R_D: Measured default tier queue occupancy, rate
2: O_A, R_A: Measured alternate tier queue occupancy, rate
3: $L_D \leftarrow \frac{O_D}{R_D}$ and $L_A \leftarrow \frac{O_A}{R_A}$
4: $p \leftarrow \frac{R_D}{R_D+R_A}$
5: **if** $L_D < L_A$ **then**
6: mode $\leftarrow$ `promotion`
7: **else**
8: mode $\leftarrow$ `demotion`
9: $\Delta p \leftarrow$ ComputeShift(p, L_D, L_A)
10: $S \leftarrow$ FindPages(mode, Δp, $min(\Delta p(R_D+R_A), M)$)
11: **if** mode is `promotion` **then**
12: promote pages in S from alternate to default
13: **else**
14: demote pages in S from default to alternate

Desired shift in per-tier access probability. In existing memory tiering systems, the decision on the set of hot pages to migrate during each quantum is trivial—simply migrate as many hot pages as possible (while respecting migration rate limits). For Colloid, however, the principle of balancing access latencies changes the structure of the problem—the equilibrium point may correspond to placing a subset of hot pages in the default tier and a subset of the hot pages in the alternate tier. As a result, the decision on the set of hot pages to migrate during each quantum is non-trivial. Migrating too many hot pages may result in oscillations around the equilibrium point. Migrating too few hot pages, may result in longer time to converge to the equilibrium point. Colloid determines the set of hot pages to migrate by computing the desired shift in per-tier access probabilities.

Algorithm 2 : Computing desired shift in access probability.

/* Initialize $p_{lo} \leftarrow 0$ and $p_{hi} \leftarrow 1$ */
1: **procedure** ComputeShift(p, L_D, L_A)
2: **if** $|L_D - L_A| < \delta \cdot L_D$ **then return** 0
3: **else**
4: **if** $L_D < L_A$ **then** $p_{lo} \leftarrow p$ **else** $p_{hi} \leftarrow p$
5: **if** $p_{hi} < p_{lo} + \epsilon$ **then**
6: **if** $L_D < L_A$ **then** $p_{hi} \leftarrow 1$ **else** $p_{lo} \leftarrow 0$
return $\left|\frac{p_{lo}+p_{hi}}{2} - p\right|$

To find the desired shift in access probability, Colloid executes a binary-search style procedure (Algorithm 2) using two watermarks, p_{lo} and p_{hi}. Intuitively, p_{hi} upper bounds the sum of access probabilities (known so far based on measurements during previous quanta) for which the default tier access latency *may be* smaller than that of alternate tier; p_{lo}, on the other hand, bounds the sum of access probabilities (known so far based on measurements during previous quanta) for which the default tier access latency is *definitely* smaller than that of alternate tier. The above intuitive definitions suggest that p_{lo} should be initialized to 0 and p_{hi} should be initialized to 1, based on the unloaded access latencies of the tiers. Moreover, if there is a feasible equilibrium point p^*, it will be always be somewhere between the low and high watermarks—that is, $p_{lo} \le p^* \le p_{hi}$. During each quantum, Colloid tries to reduce the gap between p_{lo} and p_{hi}, thus moving the system closer to the equilibrium point. If default tier access latency is smaller than that of alternate tier, then p_{lo} is updated to p, otherwise, p_{hi} is updated to p. Colloid then moves the system towards the midpoint of p_{lo} and p_{hi}. To that end, the desired shift in access probability across the tiers is computed as $\left|\frac{p_{lo}+p_{hi}}{2} - p\right|$.

As illustrated in Figure 4(a), for any given static workload, the above procedure will converge to the equilibrium point

if one exists. This is because the gap between p_{lo} and p_{hi} reduces during each quantum, while the invariants $p_{lo} \le p \le p_{hi}$ and $p_{lo} \le p^* \le p_{hi}$ continue to hold. As a result, p eventually converges to p^*. If the access latency of the default tier is lower than the alternate tier even when $p = 1$, then Colloid converges to the optimal operating point of $p = 1$.

The key challenge is to handle dynamic time-varying workloads. Such workloads can result in two unexpected changes: (1) changes in workload access patterns can result in abrupt change in p value violating the $p_{lo} \le p \le p_{hi}$ invariant; and, (2) changes in memory interconnect contention can result in abrupt change in p^* value violating the $p_{lo} \le p^* \le p_{hi}$ invariant. As illustrated in Figure 4(b), changes in p are automatically handled by the above procedure, since the watermarks are updated before computing the Δp value.

Changes in p^*, however, require more care. For example, say p^* abruptly becomes larger than p_{hi} due to change in memory interconnect contention, as illustrated at $t = 3$ in Figure 4(c). Since the above procedure ensures that p remains lower than p_{hi} at all times and since p^* is now larger than p_{hi}, p will fail to converge to the new p^*. To handle dynamic changes in the equilibrium point, Colloid checks if the watermarks have gotten very close to each other and if the system has not yet reached the equilibrium point (that is, the access latencies of the tiers are not balanced); if so, Colloid resets the watermarks based on the access latencies of the tiers. Specifically, if p_{lo} is very close to p_{hi} and the default tier access latency is smaller than that of the alternate tier, then p^* has likely exceeded p_{hi}; thus Colloid resets p_{hi} to 1. Similarly, if p_{lo} is very close to p_{hi} and default tier access latency is larger than that of alternate tier, then p^* has likely become smaller than p_{lo}; thus Colloid resets p_{lo} to 0. Resetting the watermarks when needed enables Colloid to converge to the equilibrium point even when the equilibrium point shifts due to changes in memory interconnect contention. This is illustrated in Figure 4(c). To quantify the gap between the watermarks, we use a threshold ϵ ($0 < \epsilon < 1$), and to determine whether the system is close to the equilibrium point we check whether the access latencies of the tiers are within a factor δ of each other ($0 < \delta < 1$). Given fixed δ, increasing ϵ leads to faster detection of dynamic workload changes at the cost of worse stability. Given fixed ϵ, increasing δ leads to better stability at the cost of suboptimal steady-state throughput.

Dynamic migration limit. As Colloid converges closer to the equilibrium point, it must carefully navigate the trade-off between the desired shift in access probability Δp and the impact of additional memory traffic generated by page migrations on application performance. During a given quantum, if Δp is small and there are many pages each with tiny access probability (*e.g.*, if $\Delta p = 0.01$, and there are 1000 pages with probability 0.00001 each), Colloid may end up unnecessarily migrating a large number of pages. This may lead to a large volume of migration traffic, causing oscillations around the equilibrium point, and hurting application performance. To address this, Colloid introduces a dynamic migration limit proportional to Δp. If migration traffic is larger than the desired perturbation in rates $\Delta p(R_D + R_A)$, then it is not ideal to execute such a migration. Therefore, Colloid sets the migration limit to the minimum of $\Delta p(R_D + R_A)$ and a static limit based on maximum rate-limit for migration traffic, which is a configurable parameter in existing systems.

Selecting pages to migrate. Given a desired shift in access probability Δp, Colloid uses the access tracking mechanism in the underlying system to find a small set of pages that meet the bound on the desired shift in access probability (§4).

4 Colloid with existing memory tiering systems

Colloid leverages existing access tracking and page migration mechanisms, thus enabling easy integration with existing memory tiering systems. We integrate Colloid with three state-of-the-art memory tiering systems, HeMem [48], MEMTIS [29] and TPP [35]. This section provides the most interesting system-specific implementation details.

Integrating Colloid with any system requires implementing access latency measurement (§3.1), Colloid page placement algorithm (§3.2), and a page finding procedure based on available access tracking information. Colloid implementation uses the same trigger for page placement decisions (*e.g.*, periodically, or on page faults), and the same page migration limits as the underlying system.

4.1 HeMem with Colloid

HeMem uses Processor Event-Based Sampling (PEBS) to compute per-page access frequencies. In particular, it uses a busy polling thread that reads PEBS access samples with a fixed sampling rate, and updates per-page frequency counts. It maintains a hot list and cold list of pages for each tier. Pages are placed in the hot list if their access frequency count exceeds a fixed threshold. Pages are cooled by halving their frequency count whenever the frequency count of any page reaches a certain threshold (`COOLING_THRESHOLD`). Page migrations are performed asynchronously by a migration thread using a quantum of 10ms.

We implement Colloid on top of HeMem with 520 lines of code. Colloid access latency measurements are performed on HeMem's migration thread—CHA hardware counters are sampled every quantum to obtain average queue occupancy and average rate of requests for each tier. Colloid page placement algorithm, that replaces the page placement algorithm of HeMem, is also implemented on the migration thread. We leverage HeMem's per-page frequency counts to compute the access probability of each page. Specifically, the access probability of each page is obtained by dividing its frequency count by the cumulative frequency count across all pages. To efficiently identify pages that correspond to Δp access probability, we extend HeMem's hot/cold lists:

rather than binary hot/cold lists, we split the frequency space (0-COOLING_THRESHOLD) into equal sized bins and maintain a separate page list per bin. We use 5 bins by default. These lists are updated in the exact same fashion as HeMem's original hot/cold lists. Using these, we can easily determine which pages to migrate: we iterate over bins to find pages whose sum of access probability is less than or equal to Δp, until either Δp is satisfied, migration limit is hit, or there are no feasible page choices.

4.2 MEMTIS with Colloid

MEMTIS is similar to HeMem, with four key differences. First, it uses a dynamic sampling rate for PEBS to reduce CPU overhead. Second, it uses a dynamic threshold (based on the measured access distribution) to determine the hot and the cold page list for each tier. Third, it performs promotion and demotion using separate per-tier `kmigraterd` threads using a quantum of 500ms. Finally, MEMTIS performs page size determination—based on certain heuristics, MEMTIS coalesces pages into hugepages using a background thread, and splits hugepages into pages using `kmigraterd` threads.

We implement Colloid on top of MEMTIS with 411 lines of code. Since Colloid design is agnostic to the sampling rate used to maintain per-page frequency counts, our implementation on top of MEMTIS computes per-page access probabilities similar to our implementation on top of HeMem. We do not modify MEMTIS hot/cold list management, and simply use the per-tier hot lists to select pages for migration. We implement Colloid latency measurement and page placement algorithm on the alternate tier `kmigraterd` thread (default tier `kmigraterd` is unmodified and demotes cold pages based on capacity constraints as before). To determine which pages to migrate, we scan the corresponding tier's hot list and pick pages until either Δp is satisfied or the migration limit is hit. To handle different page sizes, we simply need to take each individual page's size into account when checking the migration limit.

4.3 TPP with Colloid

TPP periodically scans process page tables and marks pages with a special protection bit. Subsequent accesses to these pages result in a hint page fault. Hot and cold pages are determined using time-to-fault, that is, time duration between a page being marked and subsequent hint fault being triggered. A page is deemed to be hot if the time-to-fault is larger than a dynamically adapted threshold, and cold otherwise. Upon hint fault for a page in the alternate tier, it is synchronously promoted to the default tier if the page is deemed to be hot. Demotion of cold pages from default tier to alternate tier happens asynchronously through the `kswapd` thread which demotes pages based on capacity watermarks. Cold pages for demotion are picked from the kernel's inactive list (which is maintained based on page access bits).

We implement Colloid on top of TPP in Linux kernel v6.3 with ~315 lines of code. We implement Colloid access latency measurement in a kernel module that runs a spin polling thread which samples CHA counters at microsecond-scale, and exposes occupancy and rate metrics to the core kernel. This requires dedicating one core similar to vanilla HeMem. To compute per-page access probability, we use time-to-fault as a proxy—pages with higher access probability are likely to hint fault more quickly than those with lower access probability. Specifically, in a stream of requests, the average number of requests before a page with access probability p is accessed is $\frac{1}{p}$. If r is the current rate of requests to the corresponding tier, we get an average inter-request time of $\frac{1}{r}$. Thus, the expected time before a page is accessed (or equivalently, the time-to-fault for a page) is given by $\Delta t = \frac{1}{p \cdot r}$. Rearranging the equation, the access probability of a page can be calculated as $p = \frac{1}{\Delta t \cdot r}$.

Colloid page placement algorithm is implemented by modifying the hint fault handler. Upon hint fault of page in alternate tier, the page is promoted to default tier if current access latency of alternate tier is larger than default tier and the page's access probability is less than or equal to Δp (Algorithm 1). To enable demotion of hot pages from the default tier to the alternate tier, we enable hint faults on the default tier pages. Upon hint fault of default tier page, the page is demoted to alternate tier if default tier access latency is larger than alternate tier access latency, and its access probability is less than or equal to Δp. Demotion of cold pages from default tier to alternate tier via `kswapd` continues as before to meet capacity constraints.

5 Evaluation

We now evaluate the performance of three state-of-the-art memory tiering systems—HeMem, MEMTIS and TPP—with and without Colloid. Unless specified otherwise, all parameters of all systems are set to their default values. We evaluate TPP both with and without Transparent Huge Pages (THP); we present results with THP enabled (results with THP disabled are provided in [57]). For Colloid, we set $\epsilon = 0.01$ and $\delta = 0.05$; sensitivity analysis with these parameters is presented in [57]. Throughout, we use the same setup as in §2.

We focus on two key metrics: steady-state application throughput, and convergence time upon changes in workload and memory interconnect contention intensity. For real applications, we measure application-specific performance metrics that are described inline.

5.1 Steady-state Throughput

Figure 5 shows the steady-state throughput achieved by each system with and without Colloid, along with the best-case throughput for the same GUPS workload as in Figure 1. With 0× memory interconnect contention intensity, performance with Colloid matches performance without Colloid for all systems. With increasing memory interconnect contention, Colloid provides increasingly larger benefits in terms of

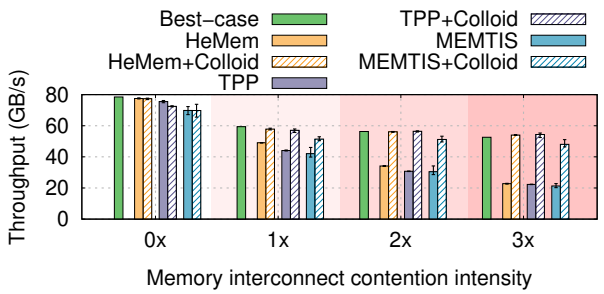

Figure 5. Colloid enables each system to achieve near-optimal performance, independent of the memory interconnect intensity. Discussion in §5.1.

steady-state throughput: 1.2−2.3× for HeMem, 1.35−2.35× for TPP and 1.29 − 2.3× for MEMTIS. Independent of the intensity of memory interconnect contention, Colloid enables HeMem, TPP and MEMTIS to achieve close to the best-case performance (within 3%, 8% and 13%, respectively).

Understanding Colloid benefits. Figure 6(a) shows that Colloid enables each system to better adapt page placement based on memory interconnect contention; intuitively, this is because Colloid page placement algorithm is able to determine the precise set of hot pages to place in each tier so as to balance access latencies as shown in Figure 6(b).

With 0× memory interconnect contention intensity, Colloid behavior is similar to the underlying system (Figure 2(b))—it places nearly the entire hot set in the default tier since the default tier access latency remains lower than that of the alternate tier. For 1×,2× and 3× memory interconnect contention intensity, unlike the results in Figure 2(b), each system now places an increasingly larger fraction of the hot set in the alternate tier, similar to the best-case placement. With 1× memory interconnect contention intensity, it balances the hot set across the tiers in order to equalize their access latencies. With 2×, and 3× memory interconnect contention intensity, Colloid places the entire hot set in the alternate tier. In these cases, even after doing so, the default tier access latency remains higher than that of the alternate tier, but the gap between the two is significantly lower compared to the Figure 2(a) measurements without Colloid for all systems.

We now perform sensitivity analysis for Colloid with varying alternate tier unloaded access latency, varying object sizes, varying number of application cores, and varying read/write ratios. Each of these parameters impact the optimal operating point in terms of the set of hot pages that must be placed in each tier to achieve optimal performance, thus enabling us to understand Colloid effectiveness across the spectrum. Here, we present results for varying alternate tier unloaded access latency and object sizes; other results are provided in [57].

Impact of alternate tier unloaded latency. Existing CXL-attached memory has 2× higher unloaded latency relative to the default tier for ASIC-based memory controllers [54, 62].

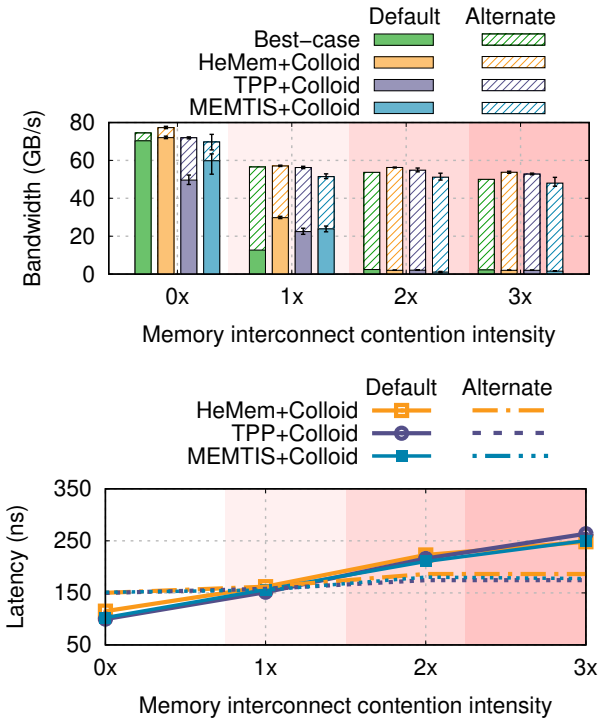

Figure 6. Understanding Colloid benefits (a-b; top-bottom) (a) Colloid enables each system to better balance hot pages across tiers based on the memory interconnect contention, similar to the best-case placement; (b) Colloid reduces the gap between the access latencies across tiers compared to Figure 2(a).

In our evaluation so far, the unloaded latency of our alternate tier is 1.9× the default tier latency, roughly matching the current CXL-attached memory latency ratio. To better understand Colloid behavior under a wider range of realistic unloaded latencies of tiered memory deployments, we increase the unloaded latency on our servers by reducing the uncore frequency of the remote socket *only for this experiment*. This enables us to vary the unloaded latency of the alternate tier from 1.9−2.7× of the default tier latency. Reducing uncore frequency has the side effect of reducing alternate tier bandwidth in addition to increasing unloaded latency. Such bandwidth reduction only hurts Colloid (that is, the reported gains are conservative)—for real hardware with comparable unloaded latency, higher bandwidth would allow Colloid to achieve better results: it will be able to place larger number of hot pages in the alternate tier since loaded latency will not increase as quickly due to higher bandwidth of the alternate tier.

Figure 7 demonstrates that Colloid enables significant performance benefits for existing memory tiering systems even when the alternate tier unloaded latency is as high as 2.7× that of the default tier. As expected, Colloid enables larger benefits for higher memory interconnect contention intensities. For a fixed memory interconnect contention intensity, the benefits of Colloid reduce with increasing alternate tier unloaded latency because the throughput

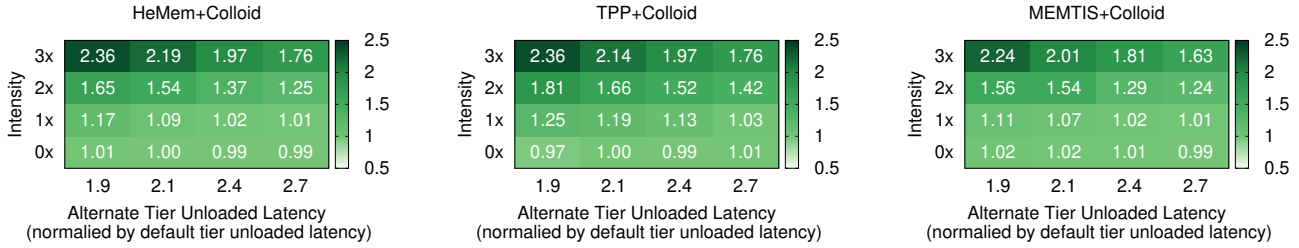

Figure 7. Colloid enables performance benefits for the underlying system even with larger alternate tier unloaded latency (a-c; left-right). Each cell in the heatmap shows performance improvement enabled by Colloid (throughput of the underlying system with Colloid normalized by throughput without Colloid) for a specific alternate tier unloaded latency and memory interconnect contention intensity.

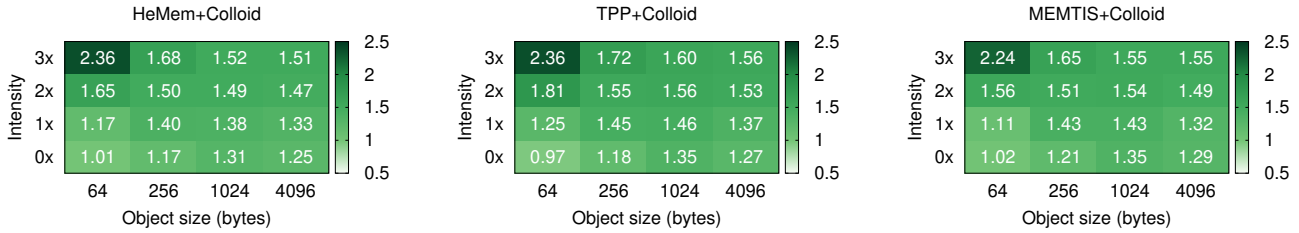

Figure 8. Colloid enables performance benefits for the underlying system, independent of the object size. With larger object sizes, Colloid enables performance improvements even at low memory interconnect contention intensity. (a-c; left-right). Each cell in the heatmap shows performance improvement enabled by Colloid (throughput of the underlying system with Colloid normalized by throughput without Colloid) for a specific object size and memory interconnect contention intensity.

benefits achieved from placing hot pages in the alternate tier reduce with higher access latency; in addition, a relatively smaller number of hot pages can be placed in the alternate tier before its access latency exceeds that of the default tier. Nevertheless, even at the maximum alternate tier unloaded latency (2.7× of default tier), Colloid still achieves $1.01-1.76\times$, $1.03-1.76\times$ and $1.01-1.63\times$ performance improvement for HeMem, TPP and MEMTIS, respectively.

Impact of object size. Figure 8 shows the benefits of Colloid with the GUPS object size varying from 64 to 4096 bytes, with all other parameters fixed to their default values. For object sizes 256 bytes and above, Colloid provides performance improvements even for 0× memory interconnect contention intensity ($1.17-1.31\times$ for HeMem, $1.18-1.35\times$ for TPP and $1.21-1.35\times$ for MEMTIS). This is because larger object sizes make the workload access pattern more sequential; as a result, hardware prefetchers perform better and the effective per-core parallelism for memory objects in increased. For example, with HeMem at 0× intensity, the average number of in-flight L3 misses per core at the CHA—which corresponds to effective per-core parallelism—are 2.82× higher for 4096 byte object size compared to 64 byte object size. Thus, the application becomes more memory intensive, causing the default tier access latency to exceed that of the alternate tier even at 0× intensity. For example, with HeMem at 0× intensity, the default tier access latency exceeds the alternate tier latency by 1.77× for 4096 byte object size enabling Colloid to provide performance improvement even at 0× intensity.

With higher memory interconnect contention intensity, Colloid benefits reduce a little bit with increase in object sizes. This is because the alternate tier memory interconnect becomes contended, thus limiting the additional throughput that Colloid can achieve by placing hot pages in the alternate tier. For example, with HeMem+Colloid at 3× intensity and for 64 byte object size, the alternate tier bandwidth utilization is 53% of the theoretical maximum bandwidth for this workload; for 4096 byte object size, the alternate tier bandwidth utilization is 96% of the theoretical maximum bandwidth for this workload.

Colloid CPU overheads. Colloid requires additional CPU cycles for latency measurements and page placement algorithm. Measurements for Figure 5 experiments for each system without and with Colloid suggest that Colloid has <2% CPU overheads for HeMem and MEMTIS, independent of the memory interconnect contention intensity. Colloid has slightly higher CPU overheads ($4-6.5\%$) for TPP due to an additional core being used for access latency measurement (§4).

5.2 Convergence Time

Adapting to dynamic, time-varying, workloads is a common challenge for any memory tiering system—if the workload changes faster than the tiering system can adapt, then the effectiveness of the memory tiering system reduces. Fundamentally, there are two sources of dynamism: change in memory access pattern, and change in memory interconnect contention. We evaluate the impact of Colloid on the convergence time of the underlying system to each of these separately.

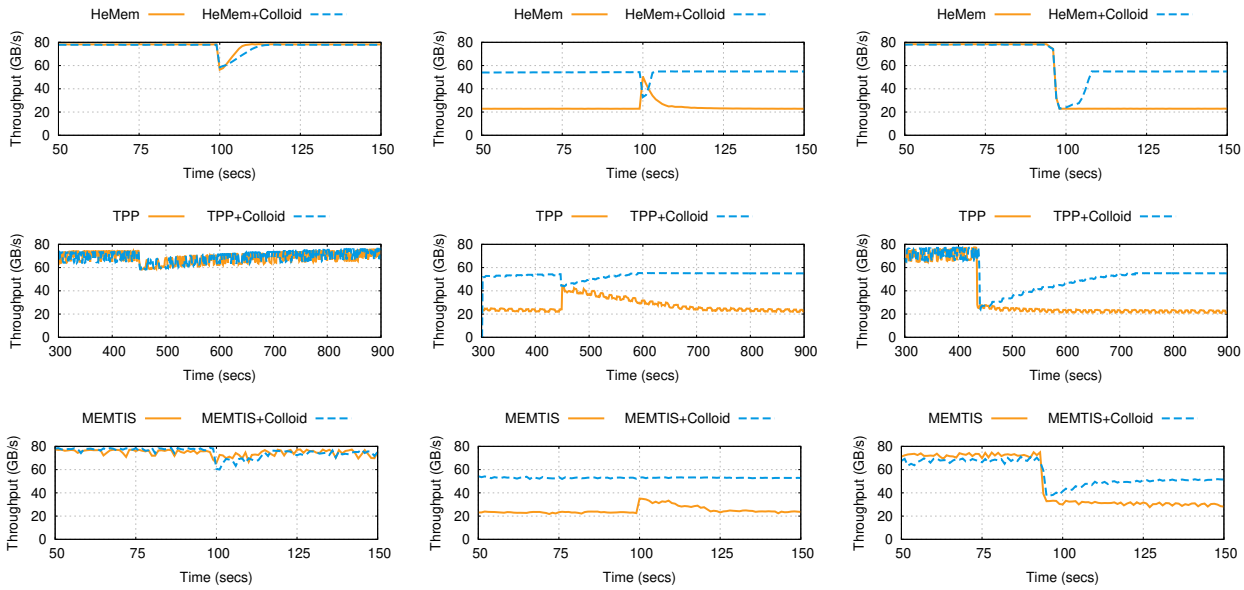

Figure 9. Colloid enables the underlying system to maintain its convergence time upon dynamic changes in access pattern and/or memory interconnect contention intensity. Each row (top-bottom) shows instantaneous throughput for the system, measured at per-second timescale, with and without Colloid. Each column uses a different scenario: (left) change in access pattern under no memory interconnect contention (center) change in access pattern under memory interconnect contention (right) change in memory interconnect contention intensity.

Dynamism due to change in access pattern. We dynamically change the hot set of the GUPS workload using the same methodology as in HeMem [48]. We run the application on 15 cores with 0× memory interconnect contention intensity and allow enough time for each of the systems to reach steady-state. Then, at a particular instant of time ($t = 100$), we instantaneously change the workload's hot set—pages previously in the hot set are marked as cold, and a different set of random pages are marked as hot. Figure 9 (top, left) shows the result with HeMem. Before the change, both HeMem and HeMem+Colloid are operating at the same throughput—both of them place the entire hot set in default tier. Upon the change, there is a sudden drop in throughput since several pages in the new hot set are now in the alternate tier. Both HeMem and HeMem+Colloid detect the change in access distribution, migrate the new hot pages to default tier, and converge back to the original throughput. We observe similar trends for TPP and MEMTIS, as shown in Figure 9 (left, center) and (left, bottom). TPP takes significantly longer relative to HeMem and MEMTIS (100s of seconds) to converge after change in access pattern because it takes longer to identify the new hot pages due to page table scan latency and less precise access information; Colloid does not change TPP's convergence times.

Figure 10 (left) shows the migration rate (that is, the number of bytes migrated per second) for HeMem and HeMem+Colloid over time for the above experiment. As expected, both HeMem and HeMem+Colloid observe sudden increase in migration rate upon change of the workload. However, compared to HeMem, HeMem+Colloid migration rate decreases more gradually; this is because of the dynamic migration limit in the Colloid algorithm (§3.2)—as the system approaches the equilibrium point, smaller Δp values lead to lower migration rates. Overall, this relatively more gradual decrease in migration rate results in HeMem+Colloid takes marginally longer than HeMem (~3 seconds) to converge. This overhead does not scale with increasing convergence time; for example, if the hot set size is larger, HeMem+Colloid migration rate will be bound by the static migration limit for most of the time; the dynamic migration limit will apply only when Δp becomes relatively small at which point Colloid is already close to convergence. Overall, HeMem+Colloid does not exceed HeMem's peak migration rate. In the steady state, HeMem+Colloid has marginally higher migration rate compared to HeMem, but this does not impact application performance since it is very small (<0.7% of application throughput). For TPP and MEMTIS peak migration rate is <1.6% and 4.5% of application throughput, respectively (results in [57]).

We repeat the same experiment as above, but with 3× memory interconnect contention intensity, to understand the behavior of the systems with change in access distribution under memory interconnect contention. Figure 9 (top, center) shows that, before the change, HeMem+Colloid operates at higher throughput by placing part of the hot set in the alternate tier. Upon the change, HeMem throughput increases since pages in the new hot set are now in the alternate tier; HeMem detects the change and migrates all hot pages back to the default tier thus converging back to its original suboptimal throughput. On the other hand, upon the change, HeMem+Colloid throughput degrades since larger-than-ideal fraction of the new hot set is in the default

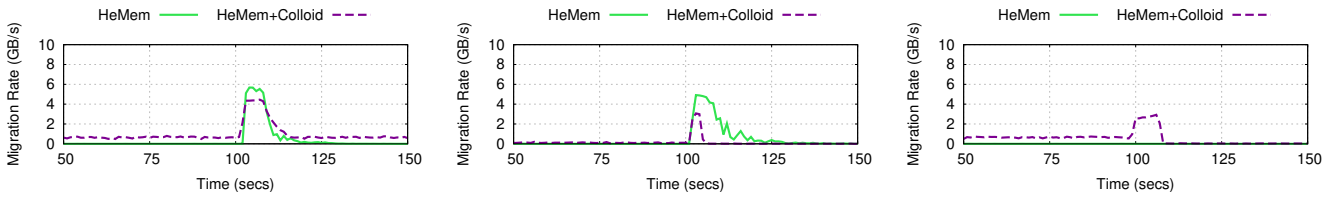

Figure 10. Migration rate (measured per-second) over time for HeMem and HeMem+Colloid corresponding to Figure 9 experiments.

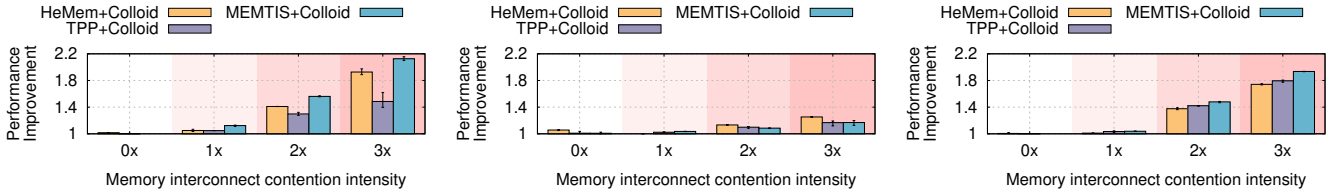

Figure 11. Colloid enables end-to-end performance benefits for real-world applications that exhibit different compute-to-memory bandwidth demands and access patterns (a-c; left-right): (a) GAPBS, a graph processing application; (b) Silo, an in-memory transactional database; and, (c) CacheLib, an in-memory key-value cache. Discussion in §5.3.

tier. HeMem+Colloid detects the change and recovers back to its original throughput by migrating pages to maintain the ideal fraction of hot set in the alternate tier. Both HeMem and HeMem+Colloid converge within 10 seconds. We observe a similar trend for TPP, although at a different timescale. MEMTIS+Colloid throughput is not impacted by the change. This is becuase MEMTIS proactively places cold pages in the alternate tier even when there is capacity available in the default tier. As a result, for MEMTIS+Colloid, the entire working set is already in the alternate tier before the change.

Dynamism due to change in memory interconnect contention. We keep the workload access distribution fixed, and dynamically change the intensity of memory interconnect contention. Initially, the application is run on 15 cores with 0× memory interconnect contention. At a particular time instant, we introduce 3× memory interconnect contention. Figure 9 (top, right) shows that, before the change, both HeMem and HeMem+Colloid operate at same throughput with the entire hot set in default tier. Upon the change, there is a sudden drop in throughput due to increased memory interconnect contention. HeMem, being agnostic to memory interconnect contention, does not react and continues to operate at degraded throughput. HeMem+Colloid detects the change, migrates pages in the hot set to alternate tier, and converges to a point of higher throughput within ∼10 seconds (similar timescale as its reaction to access pattern changes). The throughput that it converges to closely matches the best-case throughput for 3× memory interconnect contention (Figure 5). We observe similar trends for TPP and MEMTIS, as shown in Figure 9—both of them converge to near best-case throughput at similar timescales at which they adapt to access pattern changes (∼250s and ∼25s, respectively).

5.3 Real Applications

We now evaluate Colloid benefits to end-to-end performance of three real-world applications that exhibit different compute-to-memory bandwidth demands and access patterns. As before, we run each application on 15 cores and evaluate with varying intensity of memory interconnect contention.

Graph processing (GAPBS). This application implements several graph processing algorithms operating on top of in-memory graphs, producing a large number of memory accesses in the process. We evaluate the PageRank algorithm in GAPBS [5] on a Twitter graph, similar to [29]. Access locality arises from skew in the degree distribution of graph nodes. We set the default tier size to one-third of the total working set size (∼12.6GB) and execute 16 trials of the algorithm. The performance metric is the average execution time across all trials (lower is better). Figure 11(a) shows that, with increasing memory interconnect contention, Colloid improves the performance of HeMem by 1.05×−1.92×, TPP by 1.05−1.48× and MEMTIS by 1.12 − 2.12×. Performance improvement with TPP is relatively smaller compared to HeMem and MEMTIS because it takes longer to identify hot pages.

In-memory transaction processing (Silo). This application implements a high-performance in-memory transactional database; we evaluate it using the YCSB-C benchmark, as in [29]. The working set consists of 400 million key-value pairs with 64 byte keys and 100 byte values; the total working set size is thus ∼60GB. Default tier size is set to one-third of the working set size. We execute 15 billion lookup operations using a Zipfian access distribution. Figure 11(b) shows the corresponding results. Colloid matches baseline performance at low memory interconnection contention and improves the performance of HeMem by 1.13 − 1.25×, TPP by 1.09−1.17× and MEMTIS by 1.08−1.17×, at higher memory interconnection contention intensities.

In-memory key-value cache (CacheLib). We evaluate CacheLib, a popular open-source caching library, in RAM-only mode and execute workloads using the standard CacheBench tool. We use the key-value workload from [48] (HeMemKV): the key and the value sizes are fixed to 64B and 4KB, respectively, 20% of keys are in the hot set, and remaining are in the cold set. The hot set is accessed uniformly at random with 90% probability, and cold set with 10% probability. The GET/UPDATE ratio is 90/10. We populate 15 million KV pairs leading to working set size of ~75GB, set the default tier size to one-third of the working set size, and execute 1 billion operations from 15 cores. The performance metric is average throughput (operations per second) over the entire duration of the experiment. Figure 11(c) shows that, at 2× and 3× memory interconnect contention intensity, Colloid improves performance of HeMem by $1.37-1.74\times$, TPP by $1.42-1.79\times$, and MEMTIS by $1.48-1.93\times$.

6 Related Work

We discuss key works related to Colloid.

Memory management for NUMA architectures. Classical literature in this space [6, 7, 11, 30, 34, 56] has primarily focused on the problem of placing both compute tasks and memory pages in multi-socket systems to optimize application performance. The work that comes closest to our problem is Carrefour [11]. Carrefour, among other techniques, performs page placement to balance load (average rate of requests) across memory attached to different NUMA nodes. In the memory tiering context, where each tier has a different unloaded latency, this will lead to unnecessarily placing pages in alternate tiers even when the memory interconnect is not contended. Moreover, under memory interconnect contention, balancing request rates could lead to suboptimal application performance. Colloid demonstrates that memory management using the principle of balancing access latencies is a better approach to maximize application performance.

Software-managed tiered memory management. We have already demonstrated the benefits of Colloid with state-of-the-art software-based memory tiering systems [29, 35, 48]. Colloid can be integrated with any of the other existing systems [1, 9, 12, 23, 24, 60, 61] to perform memory management using the principle of balancing access latencies.

BATMAN [9] performs tiered memory management using a bandwidth-centric approach—it balances fractions of accesses to the tiers based on the ratio of their theoretical maximum bandwidths, independent of memory interconnect contention. Such an approach is suboptimal due to two reasons. First, when unloaded latencies of the tiers are different (as is the case for modern tiered memory architectures), this approach may unnecessarily place hot pages in tiers with higher access latency leading to suboptimal application performance. Second, as discussed in §3.1, access latency of tiers can increase significantly, far before their maximum bandwidth is saturated; as a result, using bandwidth as a metric fails to fully capture the impact of memory interconnect contention. Colloid automatically captures unloaded latencies and memory interconnect contention (independent of whether memory bandwidth is saturated or not) using the principle of balancing access latencies.

Hardware-managed tiered memory management. An alternate approach to tiered memory management is to use the default tier as an inclusive cache (*e.g.*, Intel memory mode [31, 48], stacked DRAM caches [15, 22, 33, 47, 50]) or exclusive cache (*e.g.*, Intel flat memory mode [62]) for the alternate tier. These approaches have the benefit of enabling data placement across tiers at cacheline granularity. However, existing hardware-managed tiered memory systems make the same assumption as their software counterparts: despite serving most of the hot data from the default tier, the access latency of the default tier remains lower than that of the alternate tier. It would be interesting to integrate Colloid with hardware-based tiered memory management mechanisms to perform data placement at cacheline granularity using the principle of balancing access latencies.

7 Conclusion

Existing memory tiering systems innovate on mechanisms for access tracking, page migration, and dynamic page size determination; however, they all use the same page placement algorithm—packing the hottest pages in the default tier (one with the lowest hardware-specified memory access latency). This is based on the implicit assumption that, despite serving the hottest pages, access latency of the default tier is always lower than that of alternate tiers. We have demonstrated that this assumption does not hold in the regime of multiple concurrent memory requests leading to memory interconnect contention, and that existing systems achieve far from optimal performance in this regime. We have presented Colloid, a memory management mechanism that embodies the principle that page placement should be adapted to balance the access latencies of the memory tiers. This principle provides a unified approach to memory management, enabling Colloid to optimize application performance independent of the latency and bandwidth characteristics of individual memory tiers, with and without memory interconnect contention. We have integrated Colloid with three state-of-the-art memory tiering systems, and demonstrated its effectiveness over a wide variety of workloads and real-world applications.

Acknowledgements

We would like to thank our shepherd, Sudarsun Kannan, the SOSP reviewers, Qizhe Cai, Shouxu Lin, Shreyas Kharbanda and Benny Rubin for their insightful feedback. This research was in part supported by NSF grant CNS-1704742, and a Sloan fellowship.

References

[1] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *ACM ASPLOS*.

[2] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, and Amin Vahdat. 2022. Understanding Host Interconnect Congestion. In *ACM HotNets*.

[3] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. 2023. Host Congestion Control. In *ACM SIGCOMM*.

[4] AMD. 2024. Performance Monitor Counters for AMD Family 1Ah Model 00h0Fh Processors. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/58550-0.01.pdf.

[5] Scott Beamer. 2023. GAPBS PageRank Implementation. https://github.com/sbeamer/gapbs/blob/master/src/pr.cc.

[6] William Bolosky, Robert Fitzgerald, and Michael Scott. 1989. Simple But Effective Techniques for NUMA Memory Management. In *ACM SOSP*.

[7] Timothy Brecht. 1993. On The Importance of Parallel Application Placement in NUMA Multiprocessors. In *USENIX SEDMS*.

[8] Shuang Chen, Christina Delimitrou, and José F Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *ACM ASPLOS*.

[9] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. 2017. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM. In *ACM MEMSYS*.

[10] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. In *ACM CSUR*.

[11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *ACM ASPLOS*.

[12] 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. 2023. Towards an Adaptable Aystems Architecture for Memory Tiering at Warehouse-Scale. In *ACM ASPLOS*.

[13] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. 2010. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *ACM ASPLOS*.

[14] Saugata Ghose, Hyodong Lee, and Jose F Martinez. 2013. Improving Memory Scheduling via Processor-Side Load Criticality Information. In *IEEE/ACM ISCA*.

[15] Nagendra Gulur, Mahesh Mehendale, R Manikantan, and R Govindarajan. 2014. Bi-Modal DRAM Cache: Improving Hit Rate, Hit Latency and Bandwidth. In *IEEE/ACM MICRO*.

[16] Ying Huang. 2022. [PATCH -V4 0/3] Memory Tiering: Hot Page Selection. https://lwn.net/ml/linux-kernel/20220622083519.708236-1-ying.huang@intel.com/.

[17] Intel. 2017. Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring. https://kib.kiev.ua/x86docs/Intel/PerfMon/336274-001.pdf.

[18] Intel. 2021. 3rd Gen Intel Xeon Processor Scalable Family, Codename Ice Lake, Uncore Performance Monitoring. https://cdrdv2-public.intel.com/679093/639778%20ICX%20UPG%20v1.pdf.

[19] Intel. 2024. Intel Xeon CPU Max Series. https://www.intel.com/content/www/us/en/products/details/processors/xeon/max-series.html.

[20] Intel. 2024. Sapphire Rapids (SPR) Uncore Events. https://github.com/intel/perfmon/blob/main/SPR/events/sapphirerapids_uncore_experimental.json.

[21] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. 2007. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *ACM SIGMETRICS*.

[22] Djordje Jevdjic, Gabriel H Loh, Cansu Kaynak, and Babak Falsafi. 2014. Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache. In *IEEE/ACM ISCA*.

[23] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *IEEE/ACM ISCA*.

[24] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *USENIX ATC*.

[25] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *IEEE HPCA*.

[26] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *IEEE/ACM ISCA*.

[27] Aneesh KV Kumar. 2022. [PATCH v7 00/12] mm/demotion: Memory Tiers and Demotion. https://lwn.net/ml/linux-kernel/20220622082513.467538-1-aneesh.kumar@linux.ibm.com/.

[28] Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2010. DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems. https://utw10235.utweb.utexas.edu/people/cjlee/TR-HPS-2010-002.pdf.

[29] 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 *ACM SOSP*.

[30] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA systems: Asymmetry Matters. In *USENIX ATC*.

[31] Baptiste Lepers and Willy Zwaenepoel. 2023. Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems). In *USENIX OSDI*.

[32] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *ACM ASPLOS*.

[33] Gabriel H Loh and Mark D Hill. 2011. Efficiently Enabling Conventional Block Sizes for Very Large Die-Stacked DRAM Caches. In *IEEE/ACM MICRO*.

[34] Zoltan Majo and Thomas R Gross. 2011. Memory System Performance in a NUMA Multicore Multiprocessor. In *ACM SYSTOR*.

[35] 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 *ACM ASPLOS*.

[36] John McCalpin. 2023. The Evolution of Single-Core Bandwidth in Multicore Systems. https://sites.utexas.edu/jdm4372/2023/12/19/the-evolution-of-single-core-bandwidth-in-multicore-systems-update/.

[37] John D McCalpin. 2023. Bandwidth Limits in the Intel Xeon Max (Sapphire Rapids with HBM) Processors. In *ISC High Performance*.

[38] Timothy Prickett Morgan. 2020. CXL And Gen-Z Iron Out A Coherent Interconnect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/.

[39] Thomas Moscibroda and Onur Mutlu. 2008. Distributed Order Scheduling and its Application to Multi-Core DRAM Controllers. In *ACM PODC*.

[40] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. 2011. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *IEEE/ACM ISCA*.

[41] Onur Mutlu. 2013. Memory Scaling: A Systems Architecture Perspective. In *IEEE IMW*.

[42] Onur Mutlu and Thomas Moscibroda. 2007. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security*.

[43] Onur Mutlu and Thomas Moscibroda. 2007. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *IEEE/ACM MICRO*.

[44] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *IEEE/ACM ISCA*.

[45] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. 2006. Fair Queuing Memory Systems. In *IEEE/ACM MICRO*.

[46] Dylan Patel, Jeff Koch, Tanj Bennett, and Wega Chu. 2024. The Memory Wall: Past, Present, and Future of DRAM. https://www.semianalysis.com/p/the-memory-wall.

[47] Moin Qureshi and Gabriel H Loh. 2012. Fundamental Latency Trade-Offs in Architecturing DRAM Caches: Outperforming Impractical SRAM-tags With a Simple and Practical Design. In *IEEE/ACM MICRO*.

[48] 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 *ACM SOSP*.

[49] Shigeru Shiratake. 2020. Scaling and Performance Challenges of Future DRAM. In *IEEE IMW*.

[50] Jaewoong Sim, Gabriel H Loh, Hyesoon Kim, Mike OConnor, and Mithuna Thottethodi. 2012. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *IEEE/ACM MICRO*.

[51] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. 2014. The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost. In *IEEE ICCD*.

[52] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory. In *IEEE/ACM MICRO*.

[53] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. 2013. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *IEEE HPCA*.

[54] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *IEEE/ACM MICRO*.

[55] James Tuck, Luis Ceze, and Josep Torrellas. 2006. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *IEEE/ACM MICRO*.

[56] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. 1996. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *ACM ASPLOS*.

[57] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key! (Extended Version). https://github.com/host-architecture/colloid.

[58] Midhul Vuppalapati, Saksham Agarwal, Henry N Schuh, Baris Kasikci, Arvind Krishnamurthy, and Rachit Agarwal. 2024. Understanding the Host Network. In *ACM SIGCOMM*.

[59] Hao Wang, Chang-Jae Park, Gyung-su Byun, Jung Ho Ahn, and Nam Sung Kim. 2015. Alloy: Parallel-Serial Memory Channel Architecture for Single-Chip Heterogeneous Processor Systems. In *IEEE HPCA*.

[60] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *USENIX OSDI*.

[61] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *ACM ASPLOS*.

[62] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *USENIX OSDI*.