TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory

Hasan Al Maruf
University of Michigan
USA

Hao Wang
NVIDIA
USA

Abhishek Dhanotia
Meta Inc.
USA

Johannes Weiner
Meta Inc.
USA

Niket Agarwal
NVIDIA
USA

Pallab Bhattacharya
NVIDIA
USA

Chris Petersen
Meta Inc.
USA

Mosharaf Chowdhury
University of Michigan
USA

Shobhit Kanaujia
Meta Inc.
USA

Prakash Chauhan
Meta Inc.
USA

ABSTRACT

The increasing demand for memory in hyperscale applications has led to memory becoming a large portion of the overall datacenter spend. The emergence of coherent interfaces like CXL enables main memory expansion and offers an efficient solution to this problem. In such systems, the main memory can constitute different memory technologies with varied characteristics. In this paper, we characterize memory usage patterns of a wide range of datacenter applications across the server fleet of Meta. We, therefore, demonstrate the opportunities to offload colder pages to slower memory tiers for these applications. Without efficient memory management, however, such systems can significantly degrade performance.

We propose a novel OS-level application-transparent page placement mechanism (TPP) for CXL-enabled memory. TPP employs a lightweight mechanism to identify and place hot/cold pages to appropriate memory tiers. It enables a proactive page demotion from local memory to CXL-Memory. This technique ensures a memory headroom for new page allocations that are often related to request processing and tend to be short-lived and hot. At the same time, TPP can promptly promote performance-critical hot pages trapped in the slow CXL-Memory to the fast local memory, while minimizing both sampling overhead and unnecessary migrations. TPP works transparently without any application-specific knowledge and can be deployed globally as a kernel release.

We evaluate TPP with diverse memory-sensitive workloads in the production server fleet with early samples of new x86 CPUs with CXL 1.1 support. TPP makes a tiered memory system performant as an ideal baseline (<1% gap) that has all the memory in the local tier. It is 18% better than today's Linux, and 5–17% better than existing solutions including NUMA Balancing and AutoTiering. Most of the TPP patches have been merged in the Linux v5.18 release while the remaining ones are just pending for more discussion.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; **Memory management**; • **Hardware** → **Emerging architectures**; **Memory and dense storage**.

KEYWORDS

Datacenters, Operating Systems, Memory Management, Tiered-Memory, CXL-Memory, Heterogeneous System

1 INTRODUCTION

The surge in memory needs for datacenter applications [12, 61], combined with the increasing DRAM cost and technology scaling challenges [49, 54] has led to memory becoming a significant infrastructure expense in hyperscale datacenters. Non-DRAM memory technologies provide an opportunity to alleviate this problem by building tiered memory subsystems and adding higher memory capacity at a cheaper $/GB point [5, 19, 38, 39, 46]. These technologies, however, have much higher latency vs. main memory and can significantly degrade performance when data is inefficiently placed in different levels of the memory hierarchy. Additionally, prior knowledge of application behavior and careful application

Figure 1: CXL decouples memory from compute.

Figure 2: Latency characteristics of memory technologies.
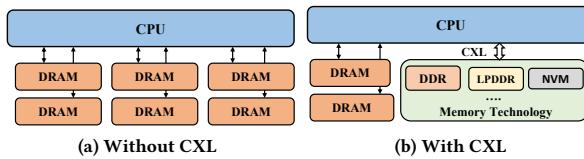
tuning is required to effectively use these technologies. This can be prohibitively resource-intensive in hyperscale environments with varieties of rapidly evolving applications.

Compute Express Link (CXL) [7] mitigates this problem by providing an intermediate latency operating point with DRAM-like bandwidth and cache-line granular access semantics. CXL protocol allows a new memory bus interface to attach memory to the CPU (Figure 1). From a software perspective, CXL-Memory appears to a system as a CPU-less NUMA node where its memory characteristics (e.g., bandwidth, capacity, generation, technology, etc.) are independent of the memory directly attached to the CPU. This allows flexibility in memory subsystem design and fine-grained control over the memory bandwidth and capacity [9, 10, 24]. Additionally, as CXL-Memory appears like the main memory, it provides opportunities for transparent page placement on the appropriate memory tier. However, Linux's memory management mechanism is designed for homogeneous CPU-attached DRAM-only systems and performs poorly on CXL-Memory system. In such a system, as memory access latency varies across memory tiers (Figure 2), application performance greatly depends on the fraction of memory served from the fast memory.

To understand whether memory tiering can be beneficial, we need to understand the variety of memory access behavior in existing datacenter applications. For each application, we want to know how much of its memory remains hot, warm, and cold within a certain period and what fraction of its memory is short- vs. long-lived. Existing Idle Page Tracking (IPT) based characterization tools [11, 17, 69] do not fit the bill as they require kernel modifications that is often not possible in productions. Besides, continuous access bit sampling and profiling require excessive CPU and memory overhead. This may not scale with large working sets. Moreover, applications often have different sensitivity towards different types of memory pages (e.g., anon page, file page cache, shared memory, etc.) which existing tools do not account. To this end, we build Chameleon, a robust and lightweight user-space tool, that uses existing CPU's Precise Event-Based Sampling (PEBS) mechanism to characterize an application's memory access behavior (§3). Chameleon generates a heat-map of memory usage on different types of pages and provides insights into an application's expected performance with multiple temperature tiers.

We use Chameleon to profile a variety of large memory-bound applications across different service domains running in our production and make the following observations. **(1)** Meaningful portions of application working sets can be warm/cold. We can offload that to a slow tier memory without significant performance impact. **(2)** A large fraction of anon memory (created for a program's stack, heap, and/or calls to mmap) tends to be hotter, while a large fraction of file-backed memory tends to be relatively colder. **(3)** Page access
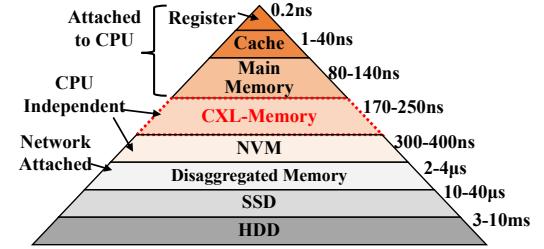
patterns remain relatively stable for meaningful time durations (minutes to hours). This is enough to observe application behavior and make page placement decisions in kernel-space. **(4)** With new (de)allocations, actual physical page addresses can change their behavior from hot to cold and vice versa fairly quickly. Static page allocations can significantly degrade performance.

Considering the above observations, we design an OS-level transparent page placement mechanism – TPP, to efficiently place pages in a tiered-memory systems so that relatively hot pages remain in fast memory tier and cold pages are moved to the slow memory tier (§5). TPP has three prime components: **(a)** a lightweight reclamation mechanism to demote colder pages to the slow tier node; **(b)** decoupling the allocation and reclamation logic for multi-NUMA systems to maintain a headroom of free pages on fast tier node; and **(c)** a reactive page promotion mechanism that efficiently identifies hot pages trapped in the slow memory tier and promote them to the fast memory tier to improve performance. We also introduce support for page type-aware allocation across the memory tiers – preferably allocate sensitive anon pages to fast tier and file caches to slow tier. With this optional application-aware setting, TPP can act from a better starting point and converge faster for applications with certain access behaviors.

We choose four production workloads that constitute significant portion of our server fleet and run them on a system that support CXL 1.1 specification (§6). We find that TPP provides the similar performance behavior of all memory served from the fast memory tier. For some workloads, this holds true even when local DRAM is only 20% of the total system memory. TPP moves all the effective hot memory to the fast memory tier and improves default Linux's performance by up to 18%. We compare TPP against NUMA Balancing [22] and AutoTiering [47], two state-of-the-art solutions for tiered memory. TPP outperforms both of them by 5–17%.

We make the following contributions in this paper:

- We present Chameleon, a lightweight user-space memory characterization tool. We use it to understand workload's memory consumption behavior and assess the scope of tiered-memory in hyperscale datacenters (§3). We open source Chameleon.

- We propose TPP for efficient memory management on a tiered-memory system (§5). We publish the source code of TPP. A major portion of it has been merged to Linux kernel v5.18. Rest of it is under an upstream discussion.

- We evaluate TPP on a CXL-enabled tiered-memory systems with real production workloads (§6) for years. TPP makes tiered memory systems as performant as an ideal system with all memory in local tier. For datacenter applications, TPP improves

default Linux's performance by up to 18%. It also outperforms NUMA Balancing and AutoTiering by 5–17%.

To the best of our knowledge, we are the first to characterize and evaluate an end-to-end practical CXL-Memory system that can be readily deployed in hyperscale datacenters.

2 MOTIVATION

Increased Memory Demand in Datacenter Applications. To build low-latency services, in-memory computation has become a norm in datacenter applications. This has led to rapid growth in memory demands across the server fleet. With new generations of CPU and DRAM technologies, memory is becoming the more prominent portion of rack-level power and total cost of ownership (TCO). (Figure 3).
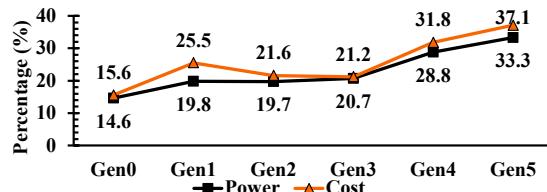
Figure 3: Memory as a percentage of rack TCO and power across different hardware generations of Meta.

Scaling Challenges in Homogeneous Server Designs. In today's server architectures, memory subsystem design is completely dependent on the underlying memory technology support in the CPUs. This has several limitations: (a) memory controllers only support a single generation of memory technology which limits mix-and-match of different technologies with different cost-per-GB and bandwidth vs. latency profiles; (b) memory capacity comes at power-of-two granularity which limits finer grain memory capacity sizing; (c) there are limited bandwidth vs. capacity points per DRAM generation (Figure 4). This forces higher memory capacity in order to get more bandwidth on the system. Such tight coupling between CPU and memory subsystem restricts the flexibility in designing efficient memory hierarchies and leads to stranded compute, network, and/or memory resources. Prior bus interfaces that allow memory expansion are also proprietary to some extent [3, 18, 44] and not commonly supported across all the CPUs [6, 14, 23]. Besides, high latency characteristics and lack of coherency limit their viability in hyperscalers.

CXL for Designing Tiered-Memory Systems. CXL [7] is an open, industry-supported interconnect based on the PCI Express (PCIe) interface. It enables high-speed, low latency communication between the host processor and devices (e.g., accelerators, memory buffers, smart I/O devices, etc.) while expanding memory capacity and bandwidth. CXL provides byte addressable memory in the same physical address space and allows transparent memory allocation using standard memory allocation APIs. It allows cache-line granularity access to the connected devices and underlying hardware maintains coherency and consistency. With PCIe 5.0, CPU to CXL interconnect bandwidth will be similar to the cross-socket interconnects (Figure 5) on a dual-socket machine. CXL-Memory access latency is also similar to the NUMA access latency. CXL adds
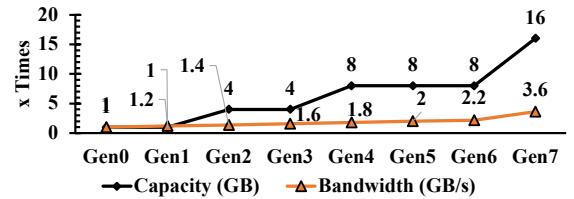
Figure 4: Memory bandwidth and capacity increase over time.

around 50-100 nanoseconds of extra latency over normal DRAM access. This NUMA-like behavior with main memory-like access semantics makes CXL-Memory a good candidate for the slow-tier in datacenter memory hierarchies.

CXL solutions are being developed and incorporated by leading chip providers [1, 4, 9, 21, 24, 25]. All the tools, drivers, and OS changes required to support CXL are open sourced so that anyone can contribute and benefit directly without relying on single supplier solutions. CXL relaxes most of the memory subsystem limitations mentioned earlier. It enables flexible memory subsystem designs with desired memory bandwidth, capacity, and cost-per-GB ratio based on workload demands. This helps scale compute and memory resources independently and ensure a better utilization of stranded resources.

Scope of CXL-Based Tiered-Memory Systems. Datacenter workloads rarely use all of the memory all the time [2, 15, 48, 70, 73]. Often an application allocates a large amount of memory but accesses it infrequently [48, 56]. We characterize four popular applications in our production server fleet and find that 55-80% of an application's allocated memory remains idle within any two minutes interval (§3.2). Moving this cold memory to a slower memory tier can create space for more hot memory pages to operate on the fast memory tier and improve application-level performance. Besides, it also allows reducing TCO by flexible server design with smaller fast memory tier and larger but cheaper slow memory tier.

As CXL-attached slow memory can be of any technology (e.g., DRAM, NVM, LPDRAM, etc.), for the sake of generality, we will call a memory directly attached to a CPU as local memory and a CXL-attached memory as CXL-Memory.

Lightweight Characterization of Datacenter Applications. In a hyperscaler environment, different types of rapidly evolving applications consume a production server's resources. Applications can have different requirements, e.g., some can be extremely latency sensitive, while others can be memory bandwidth sensitive. Sensitivity towards different memory page types can also vary for different applications. To understand the scope of tiered-memory system for existing datacenter applications, we need to characterize their memory usage pattern and quantify the opportunity of memory offloading at different memory tiers for different page types. This insight can help system admins decide on the flexible and optimized memory configurations to support different workloads.

Existing memory characterization tools fall short of providing these insights. For example, access bit-based mechanism [11, 17, 69] cannot track detailed memory access behavior because it tells whether a given page is accessed within a certain period of time. Even if a page gets multiple accesses within a tracking cycle, IPT-based tools will count that as a single access. Moreover, IPT only
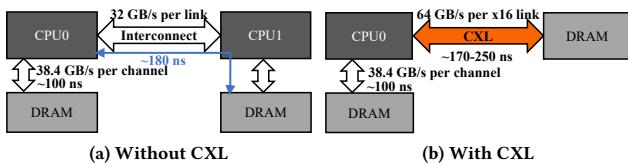
(a) Without CXL   **(b) With CXL**

Figure 5: CXL-System compared to a dual-socket server.

provides information in the physical address space – we cannot track memory allocation/deallocation if a physical page is re-used by multiple virtual pages. The overhead of such tools significantly increases with the application's memory footprint size. Even for application's with 10s of GB working set size, the overhead of IPT-based tool can be 20–90% [50]. Similarly, complex PEBS-based user-space tools [35, 63, 65, 74] lead to high CPU overheads (more than 15% per core) and often slow down the application. None of these tools generate page type-aware heat map.

3 CHARACTERIZING DATACENTER APPLICATIONS

To understand the scope of tiered-memory in hyperscale applications, we develop Chameleon, a light-weight user-space memory access behavior characterization tool. The objective of developing Chameleon is to allow users hop on any existing datacenter production machine and readily deploy it without disrupting the running application(s) and modifying the underline kernel. Chameleon's overhead needs to be comparatively lower such that it does not notably affect a production application's behavior. Chameleon's prime use case is to understand an application's memory access behavior, i.e., what fraction of an application's memory remains hot-warm-cold, how long a page survive on a specific temperature tier, how frequently they get accessed and so on. In practice, to characterize a certain type of application, we expect to run Chameleon for a few hours on a tiny fraction of servers in the whole fleet.

Considering the above objectives, we design Chameleon with two primary components – a Collector and a Worker– running simultaneously on two different threads. Collector utilizes the PEBS mechanism of modern CPUs to collect hardware-level performance events related to memory accesses. Worker uses the sampled information to generate insights.

Collector. Collector samples last level cache (LLC) misses for demand loads (event MEM_LOAD_RETIRED.L3_MISS) and optionally TLB misses for demand stores (event MEM_INST_RETIRED.STLB_MISS_STORES). Sampled records provide us with the PID and virtual memory address for the memory access events. Like any other sampling mechanism, accuracy of PEBS depends on the sampling rate – frequent samples provide higher accuracy. High sampling rate, however, incurs higher performance overhead directly on application threads and demands more CPU resources for Chameleon's Worker thread. In our fleets, sampling rate is configured as one sample for every 200 events, which appears as a good trade-off between overhead and accuracy. Note the choice of the sampling event on the store side is due to hardware limitations, i.e., there is no precise event for LLC-missed stores, likely because stores are marked complete

once TLB translation is done in modern high-end CPUs. We have conveyed the concern on this limitation to major x86 vendors in multiple occasions.

To improve flexibility, the Collector divides all CPU cores into a set of groups and enables sampling on one or more group(s) at a time (Figure 6a). After each mini_interval (by default, 5 seconds), the sampling thread rotates to the next core group. This duty-cycling helps further tune the trade-off between overhead and accuracy. It also allows sampling different events on different core groups. For example, for latency-critical applications, one can choose to sample half of the cores at a time. On the other hand, for store-heavy applications, one can enable load sampling on half of the cores and store sampling on the other half at the same time.

The Collector reads the sampling buffer and writes into one of the two hash tables. After each interval (by default, 1 minute), the Collector wakes up the Worker to process data in current hash table and moves to the other hash table for storing next interval's sampled data.

Worker. The Worker (Figure 6b) runs on a separate thread to read page access information and generate insights on memory access behavior. It considers the address of a sampled record as a virtual page access where the page size is defined by the OS. This makes it generic to systems with any page granularities (e.g., 4KB base page, 2MB huge page, etc.). To generate statistics on both virtual- and physical-spaces, the Worker finds the corresponding physical page mapped to the sampled virtual page. This address translation can cause high overhead if the target process's working set size is extremely large (e.g., terabyte-scale). One can configure the Worker to disable the physical address translation and characterize an application only on its virtual-space access pattern.

For each page, a 64-bit bitmap tracks its activeness within an interval. If a page is active within an interval, the corresponding bit is set. At the end of each interval, the bitmap is left-shifted one bit to track for a new interval. One can use multiple bits for one interval to capture the page access frequency, at the cost of supporting shorter history. After generating the statistics and reporting them, the Worker sleeps (Figure 6c).

To characterize an application deployed on hundreds of thousands of servers, we run Chameleon on a small set of servers only for a few hours. During our analysis, we chose servers where system-wide CPU and memory usage does not go above 80%. In such production environments, we do not notice any service-level performance impact while running Chameleon. CPU overhead is within 3–5% of a single core. However, on a synthetic workload that is memory bandwidth sensitive and uses all the CPU cores so that Chameleon needs to contend for CPU, we lose 7% performance due to profiling.

3.1 Production Workload Overview

We use Chameleon to characterize most popular memory-bound applications running for years across our production server fleet serving live traffic on four diverse service domains. These workloads constitute a significant portion of the server fleet and represent a wide variety of our workloads [67, 68]. **Web** implements a Virtual Machine to serve web requests. Web1 is a HipHop Virtual Machine (HHVM)-based and Web2 is a Python-based service. **Cache** is a large distributed-memory object caching service lying between the
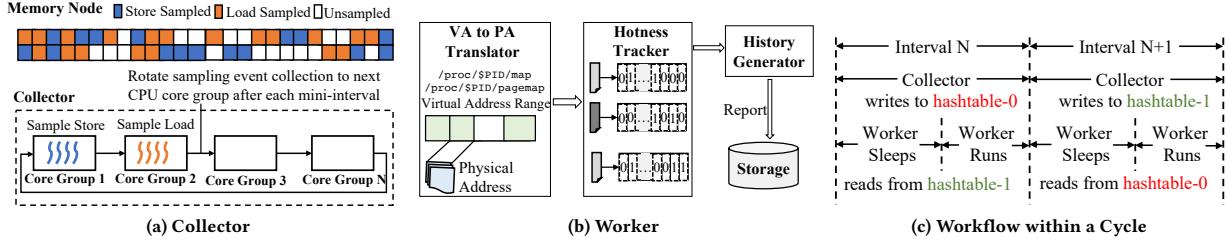
(a) Collector   **(b) Worker**   **(c) Workflow within a Cycle**

Figure 6: Overview of Chameleon components (left) and workflow (right)

Figure 7: Application memory usage over last N mins.

Figure 8: Anon pages tends to be hotter than file pages.

web and database tiers for low-latency data-retrieval. **Data Warehouse** is a unified computing engine for parallel data processing on compute clusters. This service manages and coordinates the execution of long and complex batch queries on data across a cluster. **Ads** are compute heavy workloads that retrieve in-memory data and perform machine learning computations.

3.2 Page Temperature

In datacenter applications, a significant amount of allocated memory remains cold beyond a few minutes of intervals (Figure 7). Web, Cache, and Ads use 95–98% of the system's total memory capacity, but within a two-minute interval, they use 22–80% of the total allocated memory on average.

Data Warehouse is a compute-heavy workload where a specific computation operation can span even terabytes of memory. This workload consumes almost all the available memory within a server. Even here, on average, only 20% of the accessed memory is hot within a two-minute interval.

Observation: A significant portion of a datacenter application's accessed memory remain cold for minutes. Tiered memory system can be a good fit for such cold memory if page placement mechanism can move these cold pages to a lower memory tier.

3.3 Temperature Across Different Page Types

Applications consume different types of pages based on application logic and execution demand. However, the fraction of anons (anonymous pages) remain hot is higher than the fraction of files (file pages). For Web, within a two-minute interval, 35–60% of the total allocated anons remain hot; for files, in contrast, it is only 3–14% of the total allocation (Figure 8).

Cache applications use tmpfs [27] for a fast in-memory lookup. Anons are used mostly for processing queries. As a result, file pages contribute significantly to the total hot memory. However, for Cache1, 40% of the anons get accessed within every two minutes,
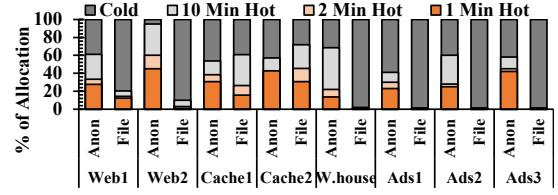
while the fraction for file is only 25%. For Cache2, the fraction of anon and file usage is almost equal within a two-minute time window. However, within a minute interval, even for Cache2, higher fraction of anons (43%) remain hot over file pages (30%).

Data Warehouse and Ads use anon pages for computation. The file pages are used for writing intermediate computation data to the storage device. As expected, almost all of hot memories are anons where almost all of the files remain cold.

Observation: A large fraction of anon pages is hot, while file pages are comparatively colder within short intervals.

Due to space constraints, we focus on a subset of these applications. In our analysis, we find the similar behavior from the rest of the applications.

3.4 Usage of Different Page Types Over Time

When the Web service starts, it loads the virtual machine's binary and bytecode files into memory. As a result, at the beginning, file caches occupy a significant portion of the memory. Overtime, anon usage slowly grows and file caches get discarded to make space for the anon pages (Figure 9a).

Cache applications mostly use file caches for in-memory lookups. As a result, file pages consume most of the allocated memory. For Cache1 and Cache2 (Figure 9b-9c), the fraction of files hovers around 70–82%. While the fraction of anon and file is almost steady, if at any point, anon usage grows, file pages are discarded to accommodate newly allocated anons.

For Data Warehouse workload, anon pages consume most of the allocated memory – 85% of the total allocated memory are anons and rest of the 15% are file pages (Figure 9d). The usage of anon and file pages mostly remains steady.

Observation: Although anon and file usage may vary over time, applications mostly maintain a steady usage pattern. Smart page placement mechanisms should be aware of page type when making placement decisions.
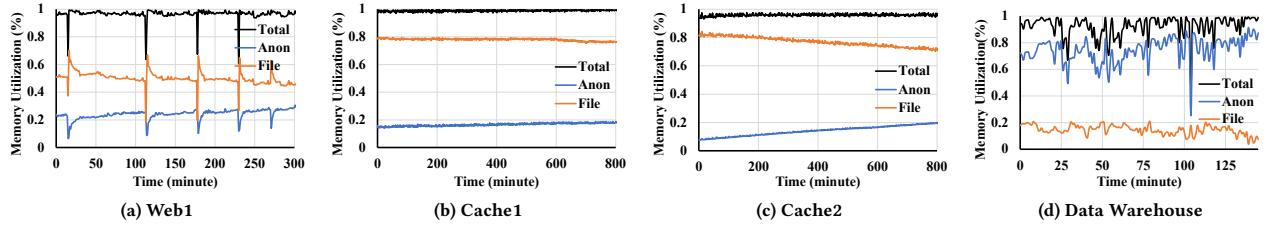
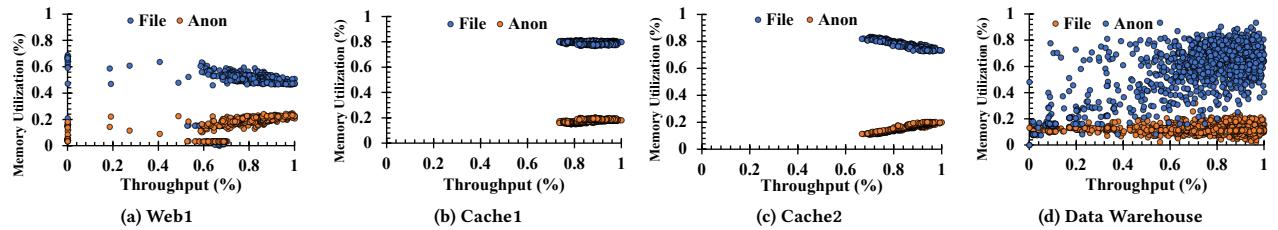**Figure 9: Memory usage over time for different applications**

Figure 10: Workloads' sensitivity towards anons and files varies. High memory capacity utilization provides high throughput.

3.5 Impact of Page Types on Performance

Figure 10 shows what fractions of different page types are used to achieve a certain application-level throughput. Memory-bound application's throughput improves with high memory utilization. However, workloads have different levels of sensitivity toward different page types. For example, Web's throughput improves with the higher utilization of anon pages (Figure 10a).

For Cache, tmpfs is allocated during initialization period. Besides, Cache1 uses a fixed amount of anons throughout its life cycle. As a result, we cannot observe any noticeable relation between anon or file usage and the application throughput (Figure 10b). However, for Cache2, we can see high throughput is achieved with comparatively higher utilization of anons (Figure 10c). Similarly, Data Warehouse application maintains a fixed amount of file pages. However, it consumes different amount of anons at different execution period and the highest throughput is achieved when the total usage of anons reaches to its highest point (Figure 10d).

Observation: Workloads have different levels of sensitivity toward different page types that varies over time.

3.6 Page Re-access Time Granularity

Cold pages often get re-accessed at a later point. Figure 11 shows the fraction of pages that become hot after remaining cold for a certain interval. For Web, almost 80% of the pages are re-accessed within a ten-minute interval. This indicates Web mostly repurposes pages allocated at an earlier time. Same goes for Cache – randomly offloading cold memory can impact performance as a good chunk of colder pages get re-accessed within a ten-minute window.

However, Data Warehouse shows different characteristics. For this workload, anons are mostly newly allocated – within a ten-minute interval, only 20% of the hot file pages are previously accessed. Rest of them are newly allocated.

Figure 11: Fraction of pages re-accessed at different intervals.

Observation: Cold page re-access time varies for workloads. Page placement on a tiered memory system should be aware of this and actively move hot pages to lower memory nodes to avoid high memory access latency.

From above observations, tiered memory subsystems can be a good fit for datacenter applications as there exists significant amount of cold memory with steady access patterns.

4 DESIGN PRINCIPLES OF TPP

With the advent of CXL technologies, hyperscalers are embracing CXL-enabled heterogeneous tiered-memory system where different memory tier has different performance characteristics [41, 52]. For performance optimization in such systems, transparent page placement mechanism (TPP) is needed to handle pages with varied hotness characteristics on appropriate temperature tiers. To design TPP for next-generation tiered-memory systems, we consider following questions:

- What is an ideal layer to implement TPP functionalities?
- How to detect page temperature?
- What abstraction to provide for accessing CXL-Memory?

In this section, we discuss the rationale and trade-offs behind design choices for TPP.

Implementation Layer. Application-transparent page placement mechanism can be employed both in the user- and kernel-space. In user-space, a Chameleon-like tool can be used to detect page temperatures and perform NUMA migrations using user-space APIs (e.g., move_pages()). To identify what to migrate, the migration tool needs to implement user-space page lists and history management. This technique entails overheads due to user-space to kernel-space context switching. It also adds processing overheads due to history management in user space. Besides, there are memory overheads due to page information management in user-space that may not scale with large working sets. While this is acceptable for profiling tools that run for short intervals on a small sample of servers in the fleet, it can be prohibitively expensive when they run continuously on all production fleet. Considering these, we design TPP as a kernel feature as we think it's less complex to implement and more performant over user-space mechanisms.

Page Temperature Detection. There are several different techniques that can potentially be used for page temperature detection including PEBS, Page Poisoning, and NUMA Balancing. PEBS can be utilized in kernel-space to detect page temperature. However, as PEBS counters are not standardized across CPU vendors, a generic precise event-based kernel implementation for page temperature detection that works across all hardware platforms is not feasible. Additionally, limited number of perf counters are supported in CPUs and are generally required to be exposed in user-space. More importantly, as mentioned earlier, even with optimizations, PEBS-based profiling is not good enough as an always-running component of TPP for high pressure workloads.

Sampling and poisoning a few pages within a memory region to track their access events is another well-established approach [16, 22, 29, 31] for finding hot/cold pages. To detect a page access, IPT-based [16, 29] approach needs to clear the page's accessed bit and flush the corresponding TLB entry. This requires monitoring accessed bits at high frequency which results in unacceptable slowdowns [29, 31]. Thermostat [31] solves this problem by sampling at 2MB page granularity which makes it effective specifically for huge-pages. One of our design goals behind TPP is that it should be agnostic to page size. In our production environment, application owners generally pre-allocate 2MB and 1GB pages and use them to allocate text regions (code), static data structures, and slab pools that serve request allocations. In most cases, these large pages are hot and should never get demoted to CXL-Memory.

NUMA Balancing (also known as AutoNUMA) [22] is transparent to OS page sizes. It generates a minor page fault when the sampled page gets accessed. Periodically incurring page faults on most frequently accessed pages can lead to high overheads. To address this, when designing TPP, we chose to only leverage minor page fault as a temperature detection mechanism for CXL-Memory. As CXL-Memory is expected to hold warm and cold pages, this will keep the overhead of temperature detection low. For cold page detection in local memory node, we find Linux's existing LRU-based age management mechanism is lightweight and quite efficient.

Empirically, we do not see any potential benefit to leverage a more sophisticated page temperature detection mechanism. In our experiments (§6), using kernel LRUs for on-the-fly profiling works well. Combining LRUs and NUMA Balancing, we can detect most

hot pages in CXL-Memory at virtually zero overhead as presented in Figure 14 and Table 1.

Memory Abstraction for CXL-Memory. One can use CXL-Memory as a swap-space to host colder memory using existing in-memory swapping mechanisms [8, 13, 26, 30]. TMO [73] is one such swap-based mechanism that detects cold pages in local memory and moves them to swap-space that is referred to as (z)swap pool. However, we do not plan to use CXL-Memory as an in-memory swap device. In such a case, we will effectively lose CXL's most important feature, i.e., load/store access semantics at cache-line granularity. With swap abstraction, every access to a (z)swapped page will incur a major page fault and we have to read the whole page from CXL-Memory. This will significantly increase the effective access latency far over 200ns and make CXL-Memory less attractive. We chose to build TPP so that applications can leverage load-store semantics when accessing warm or cold data from CXL-Memory.

While the swap semantics of TMO are not desirable in CXL-Memory page placement context, the memory saving through feedback-driven reclamation is still valuable. We think TMO as an orthogonal and complimentary tool to TPP as it operates one layer above TPP (§6.3.2). TMO runs in user-space and keeps pushing for memory reclamation, while TPP runs in kernel-space and optimizes page placement for allocated memory between local and CXL-Memory tiers.

5 TPP FOR CXL-MEMORY

An effective page placement mechanism should efficiently offload cold pages to slower CXL-Memory while aptly identify trapped hot pages in CXL-node and promote them to the fast memory tier. As CXL-Memory is CPU-less and independent of the CPU-attached memory, it should be flexible enough to support heterogeneous memory technologies with varied characteristics. Page allocation to a NUMA node should not frequently halt due to the slower reclamation mechanism to free up spaces. Besides, an effective policy should be aware of an application's sensitivity toward different page types.

Considering the datacenter workload characteristics and our design objectives, we propose TPP– a smart OS-managed mechanism for tiered-memory system. TPP places 'hotter' pages in local memory and moves 'colder' pages in CXL-Memory. TPP's design-space can be divided across four main areas – **(a)** lightweight demotion to CXL-Memory, **(b)** decoupled allocation and reclamation paths, **(c)** hot-page promotion to local nodes, and **(d)** page type-aware memory allocation.

5.1 Migration for Lightweight Reclamation

Linux tries to allocate a page to the memory node local to a CPU where the process is running. When a CPU's local memory node fills up, default reclamation pages-out to swap device. In such a case, in a NUMA system, new allocations to local node halts and takes place on CXL-node until enough pages are freed up. The slower the reclamation is, the more pages end up being allocated to the CXL-node. Besides, invoking paging events in the critical path worsens the average page access latency and impacts application performance.
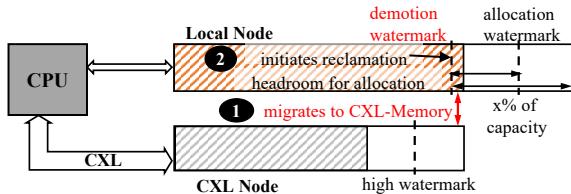
Figure 12: TPP decouples the allocation and reclamation logics for local memory node. It uses migration for demotion.

Figure 13: TPP promotes a page considering its activity state.

To enable a light-weight page reclamation for local nodes, after finding the reclamation-candidates, instead of invoking swapping mechanism, we put them in to a separate demotion list and try to migrate them to the CXL-node asynchronously (❶ in Figure 12). Migration to a NUMA node is orders of magnitude faster than swapping. We use Linux's default LRU-based mechanism to select demotion candidates. However, unlike swapping, as demoted pages are still available in-memory, along with inactive file pages, we scan inactive anon pages for reclamation candidate selection. As we start with the inactive pages, chances of hot pages being migrated to CXL-node during reclamation is very low unless the local node's capacity is smaller than the hot portion of working set size. If a migration during demotion fails (e.g., due to low memory on the CXL-node), we fall back to the default reclamation mechanism for that failed page. As allocation on CXL-node is not performance critical, CXL-nodes use the default reclamation mechanism (e.g., pages out to the swap device).

If there are multiple CXL-nodes, the demotion target is chosen based on the node distances from the CPU. Although other complex algorithms can be employed to dynamically choose the demotion target based on a CXL-node's state, this simple distance-based static mechanism turns out to be effective.

5.2 Decoupling Allocation and Reclamation

Linux maintains three watermarks (min, low, high) for each memory zone within a node. If the total number of free pages for a node goes below low_watermark, Linux considers the node is under memory pressure and initiates page reclamation for that node. In our case, TPP demotes them to CXL-node. New allocation to local node halts till the reclaimer frees up enough memory to satisfy the high_watermark. With high allocation rate, reclamation may fail to keep up as it is slower than allocation. Memory retrieved by the reclaimer may fill up soon to satisfy allocation requests. As a result, local memory allocations halt frequently, more pages end up in CXL-node which eventually degrades application performance.

In a multi-NUMA system with severe memory constraint, we should proactively maintain a reasonable amount of free memory headroom on the local node. This helps in two ways. First, new allocation bursts (that are often related to request processing and, therefore, both short-lived and hot) can be directly mapped to the local node. Second, local node can accept promotions of trapped hot pages on CXL-nodes.

To achieve that, we decouple the logic of 'reclamation stop' and 'new allocation happen' mechanism. We continue the asynchronous background reclamation process on local node until its total number of free pages reaches demotion_watermark, while new allocation
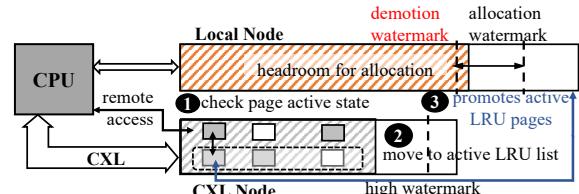
can happen if the free page count satisfies a different watermark – allocation_watermark (❷ in Figure 12). Note that demotion watermark is always set to a higher value above the allocation and low watermark so that we always reclaim more to maintain the free memory headroom.

How aggressively one needs to reclaim often depends on the application behavior and available resources. For example, if an application has high page allocation demand, but a large fraction of its memory is infrequently accessed, aggressive reclamation can help maintain free memory headroom. On the other hand, if the amount of frequently accessed pages is larger than the local node's capacity, aggressive reclamation will thrash hot memory across NUMA nodes. Considering these, to tune the aggressiveness of the reclamation process on local nodes, we provide a user-space sysctl interface (/proc/sys/vm/demote_scale_factor) to control the free memory threshold for triggering the reclamation on local nodes. By default, its value is empirically set to 2% that means reclamation starts as soon as only a 2% of the local node's capacity is available to consume. One can use workload monitoring tools [73] to dynamically adjust this value.

5.3 Page Promotion from CXL-Node

Due to increased memory pressure on local nodes, new pages may often get allocated to CXL-nodes. Besides, demoted pages may also become hot later as discussed in §3.6. Without any promotion mechanism, hot pages will always be trapped in CXL-nodes and hurt application performance. To promote such pages, we augment Linux's NUMA Balancing [22].

NUMA Balancing for CXL-Memory. In NUMA Balancing, a kernel task routinely samples a subset of a process's memory (by default, 256MB of pages) on each memory node. When a CPU accesses a sampled page, a minor page-fault is generated (known as NUMA hint fault). Pages that are accessed from a remote CPU are migrated to that CPU's local memory node (known as promotion). In a CXL-System, it is not reasonable to promote a local node's hot memory to other local or CXL-nodes. Besides, as sampling pages to find a local node's hot memory generates unnecessary NUMA hint fault overheads, we limit sampling only to CXL-nodes.

While promoting a CXL-node's trapped hot pages, we ignore the allocation watermark checking for the target local node. This creates more memory pressure to initiate the reclamation of comparatively colder pages on that node. If a system has multiple local nodes, we select the node where the task is running. When applications share multiple memory nodes, we choose local node with the lowest memory pressure.

Ping-Pong due to Opportunistic Promotion. When a NUMA hint fault happens on a page, default NUMA balancing instantly promotes the page without checking its active state. As a result, pages with very infrequent accesses can still be promoted to the local node. Once promoted, these type of pages may shortly become the demotion candidate if the local nodes are always under pressure. Thus, promotion traffic generated from infrequently accessed pages can easily fill up the local node's free space and generate a higher demotion traffic for CXL-nodes. This unnecessary traffic can negatively impact an application's performance.

Apt Identification of Trapped Hot Pages. To solve this ping-pong issue, instead of instant promotion, we check a page's age through its position in the LRU list maintained by the OS. If the faulted page is in inactive LRU, we do not consider the page for promotion instantly as it might be an infrequently accessed page. We consider the faulted page as a promotion candidate only if it is found in the active LRUs (❶ in Figure 13). This significantly reduces the promotion traffic.

However, OS uses LRU lists for reclamation. If a memory node is not under pressure and reclamation does not kick in, then pages in inactive LRU list do not automatically move to the active LRU list. As CXL-nodes may not always be under pressure, faulted pages may often be found in the inactive LRU list and, therefore, bypass the promotion filter. To address this, whenever we find a faulted page on the inactive LRU list, we mark the page as accessed and move it to the active LRU list immediately (❷ in Figure 13). If the page still remains hot during the next NUMA hint fault, it will be in the active LRU, and promoted to the local node (❸ in Figure 13).

This helps TPP add some hysteresis to page promotion. Besides, as Linux maintains separate LRU lists for anon and file pages, different page types have different promotion rates based on their respective LRU sizes and activeness. This speeds up the convergence of hot pages across memory nodes.

5.4 Page Type-Aware Allocation

The page placement mechanism we described above is generic to all page types. However, some applications can further benefit from page type-aware allocation policy. For example, production applications often perform lots of file I/O during the warm up phase and generate file caches that are accessed infrequently. As a result, cold file caches eventually end up on CXL-nodes. Not only that, local memory node being occupied by the inactive file caches often forces anons to be allocated on the CXL-nodes that may need to be promoted back later.

To resolve these unnecessary page migrations, we allow an application allocating caches (e.g., file cache, tmpfs, etc.) to the CXL-nodes preferrably, while preserving the allocation policy for anon pages. When this allocation policy is enabled, page caches generated at any point of an application's life cycle will be initially allocated to the CXL-node. If a page cache becomes hot enough to be selected as a promotion candidate, it will be eventually promoted to the local node. This policy helps applications with infrequent cache accesses run on a system with a small amount of local memory and large but cheap CXL-Memory while maintaining the performance.

Workload/Throughput (%) (normalized to Baseline)	Default Linux	TPP	NUMA Balancing	AutoTiering
Web1 (2:1)	83.5	**99.5**	82.8	87.0
Cache1 (2:1)	97.0	**99.9**	93.7	92.5
Cache1 (1:4)	86.0	**99.5**	90.0	Fails
Cache2 (2:1)	98.0	**99.6**	94.2	94.6
Cache2 (1:4)	82.0	**95.0**	78.0	Fails
Data Warehouse (2:1)	99.3	**99.5**	–	–

Table 1: TPP is effective over its counterparts. It reduces memory access latency and improves application throughput.

5.5 Observability into TPP to Assess Performance

Promotion- and demotion-related statistics can help better understand the effectiveness of the page placement mechanism and debug issues in production environments. To this end, we introduce multiple counters to track different demotion and promotion related events and make them available in the user-space through `/proc/vmstat` interface.

To characterize the demotion mechanism, we introduce counters to track the distribution of successfully demoted anon and file pages. To understand the promotion behavior, we add new counters to collect information on the numbers of sampled pages, the number of promotion attempts, and the number of successfully promoted pages for each of the memory types.

To track the ping-pong issue mentioned in §5.3, we utilize the unused `0x40` bit in the page flag to introduce `PG_demoted` flag for demoted pages. Whenever a page is demoted its `PG_demoted` bit is set and gets cleared upon promotion. We also count the number of demoted pages that become promotion candidates. High value of this counter means TPP is thrashing across NUMA nodes. We add counters for each of the promotion failure scenario (e.g., local node having low memory, abnormal page references, system-wide low memory, etc.) to reason about where and how promotion fails.

6 EVALUATION

We integrate TPP on Linux kernel v5.12. We evaluate TPP with a subset of representable production applications mentioned in §3.1 serving live traffic on tiered memory systems across our server fleet. We explore the following questions:

- How effective TPP is in distributing pages across memory tiers and improving application performance? (§6.1)
- What are the impacts of TPP components? (§6.2)
- How it performs against state-of-the-art solutions? (§6.3)

For each experiment, we use application-level throughput as the metric for performance. In addition, we use the fraction of memory accesses served from the local node as the key low-level supporting metric. We compare TPP against default Linux (§6.1), default NUMA Balancing [22], AutoTiering [47], and TMO [73] (§6.3) (Table 1). None of our experiments involve swapping to disks, the whole system has enough memory to support the workload. We use the default *local-node first, then CXL-node* allocation policy for Linux.

Experimental Setup. We deploy a number of pre-production x86 CPUs with FPGA-based CXL-Memory expansion card that support CXL 1.1 specification. Memory attached to the expansion card shows up to the OS as a CPU-less NUMA node. Although our
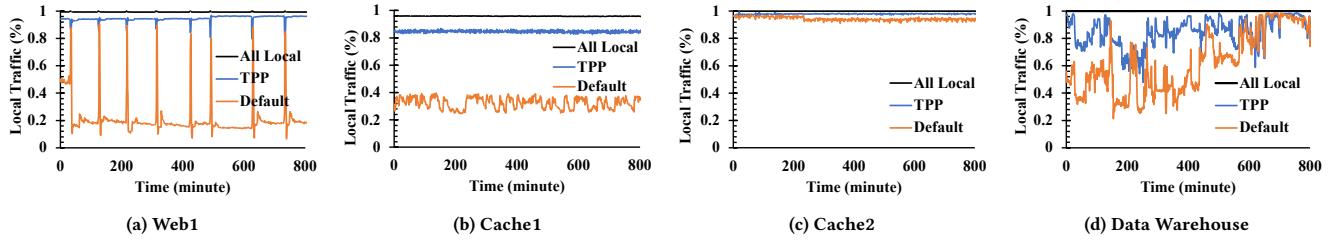
Figure 14: Fast cold page demotion and effective hot page promotion allow TPP to serve most of the traffics from the local node.

current FPGA-based CXL cards have around 250ns higher latency than our eventual target, we use them for the functional validation.

We have confirmation from two major x86 CPU vendors that the access latency to CXL-Memory is similar to the remote latency on a dual-socket system. For performance evaluation, we primarily use dual-socket systems and configure them to mimic our target CXL-enabled system's characteristics (one memory node with all active CPU cores and one CPU-less memory node) according to the guidance of our CPU vendors. For baseline, we disable the memory node and CPU cores on a socket while enabling sufficient memory on another socket. Here, single memory node serves the whole working set.

We use two memory configurations where the ratio of local node and CXL-Memory capacity is **(a)** 2:1 (§6.1.1) and **(b)** 1:4 (§6.1.2). Configuration **(a)** is similar to our current CXL-enabled production systems where local node is supposed to serve all the hot working set. We use configuration **(b)** to stress-test TPP on a constrained memory scenario – only a fraction of the total hot working set can fit on the local node and hot pages are forced to move to the CXL-node.

6.1 Effectiveness of TPP

6.1.1 Default Production Environment (2:1 Configuration)
. Web. Web1 performs lots of file I/O during initialization and fills up the local node. Default Linux is 44× slower than TPP during freeing up the local node. As a result, new allocation to the local node halts and anons get allocated to the CXL-node and stay there forever. In default Linux, on average, only 22% of total memory accesses are served from the local node (Figure 14a). As a result, throughput drops by 16.5%.

Active and faster demotion helps TPP move colder file pages to CXL-node and allow more anon pages to be allocated in the local node. Here, 92% of the total anon pages are served from the local node. As a result, local node serves 90% of total memory accesses. Throughput drop is only 0.5%.

Cache. Cache applications maintain a steady ratio of anon and file pages throughout their life-cycle. Almost all the anon pages get allocated to the local node from the beginning and served from there. For Cache1, with default Linux, only 8% of the total hot pages are trapped in CXL-node. As a result, application performance remains very close to the baseline – performance regression is only 3%. TPP can even minimize this performance gap (throughput is 99.9% of the baseline) by promoting all the trapped hot files and improving the fraction of traffic served from the local node (Figure 14b).
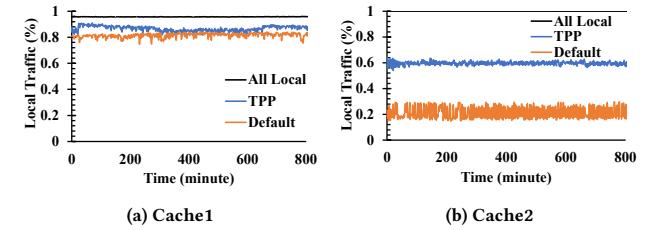

Figure 15: Effectiveness of TPP under memory constraint.

Although most of the Cache2's anon pages reside in the local node on a default Linux kernel, all of them are not always hot – only 75% of the total anon pages remain hot within a two-minute interval. TPP can efficiently detect the cold anon pages and demote them to the CXL-node. This allows the promotion of more hot file pages. On default Linux, local node serves 78% of the memory accesses (Figure 14c) and cause 2% throughput regression. TPP improves the fraction of local traffic to 91%. Throughput regression is only 0.4%.

Data Warehouse. This workload generates file caches to store the intermediate processing data. File caches mostly remain cold. Besides, only one third of the total anon pages remain hot. Our default production configuration is good enough to serve all the hot memory from the local node. Default Linux and TPP perform the same (0.5–0.7% throughput drop).

TPP improves the fraction of local traffic by moving relatively hotter anon pages to the local node. With TPP, 94% of the total anon pages reside on the local node, while the default kernel hosts only 67%. To make space for the hot anon pages, cold file pages are demoted to the CXL-node. TPP allows 4% higher local node memory accesses over Linux (Figure 14d).

6.1.2 Large Memory Expansion with CXL (1:4 Config.)
Extreme setups like 1:4 configuration allow flexible server design with DRAM as a small-sized local node and CXL-Memory as a large but cheaper memory. As in our production, such a configuration is impractical for Web and Data Warehouse, we limit our discussion to the Cache applications. Note that TPP is effective even for Web and Data Warehouse in such a setup and performs very close to the baseline.

Cache1. In a 1:4 configuration, on a default Linux kernel, file pages consume almost all the local node's capacity. 85% of the total anon pages get trapped to the CXL-node and throughput drops by 14%. Because of the apt promotion, TPP can move almost all the CXL-node's hot anon pages (97% of the total hot anon pages within
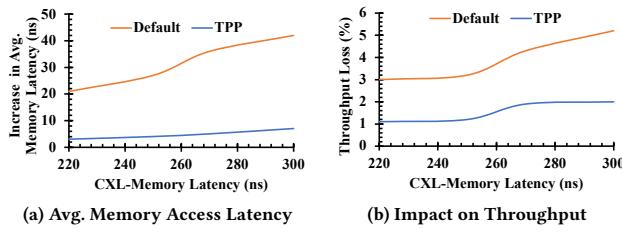
(a) Avg. Memory Access Latency

(b) Impact on Throughput

Figure 16: TPP benefits CXL-node with varied latency traits.

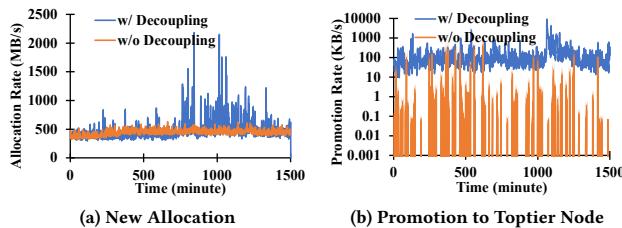**(a) New Allocation**

(b) Promotion to Toptier Node

Figure 17: Impact of decoupling allocation and reclamation.

a minute) to the local node. This forces less latency-sensitive file pages to be demoted to the CXL-node. Eventually, TPP stabilizes the traffic between the two nodes and local node serves 85% of the total memory accesses. This helps Cache1 achieve the baseline performance – even though the local node's capacity is only 20% of the working set size, throughput regression is only 0.5% (Figure 15a).

Cache2. Similar to Cache1, on default Linux, Cache2 experiences 18% throughput loss. Only 14% of the anon pages remain on the local node (Figure 15b). TPP can bring back almost all the remote hot anon pages (80% of the total anon pages) to local node while demoting the cold ones to the CXL-node. As Cache2 accesses lots of caches, and caches are now mostly in CXL-node, 41% of the memory traffic comes from the CXL-node. Yet, throughput drop is only 5% with TPP.

6.1.3 TPP with Varied CXL-Memory Latencies

. The variation in CXL-Memory's access latency does not impact TPP much. We run Cache2 with 2:1 configuration where CXL-Memory has different latency characteristics (Figure 16). In all cases, due to TPP's better hot page identification and effective promotion, only a small portion (4–5%) of hot pages are served from the CXL-node. On the other hand, for default Linux, 22–25% hot pages remain trapped in the CXL-node. This increases the average memory access latency by 7× for default Linux (Figure 16a). This, eventually, impact the application-level performance, default Linux experiences 2.2 − 2.8× higher throughput loss over TPP (Figure 16b).

6.2 Impact of TPP Components

In this section, we discuss the contribution of TPP components. As a case study, we use Cache1 with 1:4 configuration.

Allocation and Reclamation Decoupling. Without this feature, reclamation on local node triggers at a later phase. With high memory pressure and delayed reclamation on local node, the benefit of TPP disappears as it fails to promote CXL-node pages. Besides,
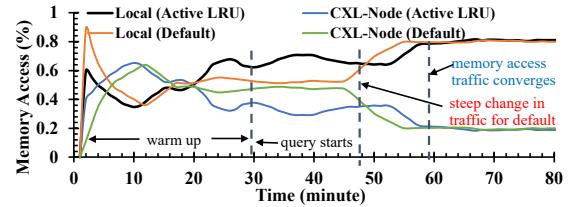

Figure 18: Restricting the promotion candidate based on their age reduces unnecessary promotion traffic.

| Application | Configuration | % of Memory Access Traffic | | Throughput |
		Local Node	CXL-node	w.r.t Baseline
Web1	2:1	97%	3%	99.5%
Cache1	1:4	85%	15%	99.8%
Cache2	1:4	72%	28%	98.5%

Table 2: Page-type aware allocation helps applications.

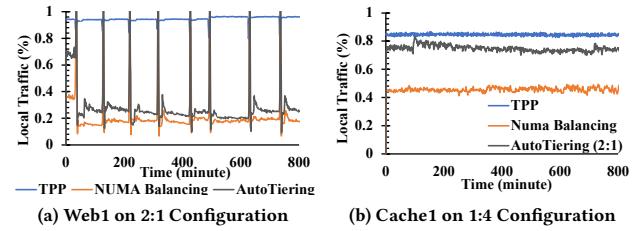**(a) Web1 on 2:1 Configuration**

(b) Cache1 on 1:4 Configuration

Figure 19: TPP outperforms existing page placement mechanism. Note that AutoTiering can't run on 1:4 configuration For Cache1, TPP on 1:4 configuration performs better than AutoTiereing on 2:1.

newly allocated pages are often short-lived (less than a minute lifetime) and de-allocated even before being selected as a promotion candidate. Trapped CXL-node hot pages worsen the performance.

Without the decoupling feature, allocation maintains a steady rate that is controlled by the rate of reclamation (Figure 17a). As a result, any burst in allocations puts pages to the CXL-node. When allocation and reclamation is decoupled, TPP allows more pages on the local node – allocation rate to local node increases by 1.6× at the 95th percentile.

As local node is always under memory pressure, new allocations consume freed up pages instantly and promotion fails as the target node becomes low on memory after serving allocation requests. For this reason, without the decoupling feature, promotion almost halts most of the time (Figure 17b). Trapped pages on the CXL-node generates 55% of the memory traffic which leads to a 12% throughput drop. With the decoupling feature, promotion maintains a steady rate of 50KB/s on average. During the surge in remote memory usage, the promotion goes as high as 1.2MB/s in the 99th percentile. This helps TPP to maintain the throughput of baseline.

Active LRU-Based Hot Page Detection. Considering active LRU pages as promotion candidates helps TPP add hysteresis to page promotion. This reduces the page promotion rate by 11×. The number of demoted pages that subsequently get promoted is also reduced by 50%. Although the demotion rate drops by 4%, as we are not allowing unnecessary promotion traffics to waste local memory node, now there are more effective free pages in local node. As a

Web1 on 2:1 Configuration	TPP-only	TPP with TMO
Migration Failure Rate (pages/sec)	20	5
CXL-node's Memory Traffic (%)	3.1%	2.7%

Table 3: TMO enhances TPP's memory reclamation process and improves page migration by generating more free space.

Web1 on 2:1 Configuration	TMO-only	TMO with TPP
Process Stall (Normalized to Threshold)	70%	40%
Memory Saving (% of Total Capacity)	13.5%	16.5%

Table 4: TPP improves TMO by effectively turning the swap action into a two-stage demote-then-swap process.

result, promotion success rate improves by 48%. Thus, reduced but successful promotions provide enough free spaces in local node for new allocations and improve the local node's memory access by 4%. Throughput also improves by 2.4%. The time requires to converge the traffic across memory tiers is almost similar – to reach the peak traffic on local node, TPP with active LRU-based promotion takes extra five minutes (Figure 18).

Cache Allocation to Remote Node Policy. For Web and Cache, preferring the file cache allocation to CXL-node can provide all-local performances even with a small-sized local node (Table 2). TPP is efficient enough to keep most of the effective hot pages on the local node. Throughput drop over the baseline is only 0.2–2.5%.

6.3 Comparison Against Existing Solutions

We compare TPP against Linux's default NUMA Balancing, AutoTiering, and TMO. We use Web1 and Cache1, two representative workloads of two different service domains, and evaluate them on target production setup (2:1 configuration) and memory-expansion setup (1:4 configuration), respectively. We omit Data Warehouse as it does not show any significant performance drop even with default Linux (§6.1).

6.3.1 TPP vs. NUMA Balancing and AutoTiering

. Web1. NUMA Balancing helps when the reclaim mechanism can provide with enough free pages for promotion. When the local node is low on memory, NUMA Balancing stops promoting pages and performs even worse than default Linux because of its extra scanning and failed promotion tries. Due to 42× slower reclamation rate than TPP, NUMA Balancing experiences 11× slower promotion rate. Local node can serve only 20% of the memory traffic (Figure 19a). As a result, throughput drops by 17.2%. Due to the unnecessary sampling, its system-wide CPU overhead is 2% higher that TPP.

AutoTiering has a faster reclamation mechanism – it migrates pages with low access frequencies to CXL-node. However, with a tightly-coupled allocation-reclamation path, it maintains a fixed-size buffer to support promotion under pressure. This reserved buffer eventually fills up during a surge in CXL-node page accesses. At that point, AutoTiering also fails to promote pages and end up serving 70% of the traffic from the CXL-node. Throughput drops by 13% from the baseline.

Note that TPP experiences only a 0.5% throughput drop.

Cache1. In 1:4 configuration, with more memory pressure on local node, NUMA Balancing effectively stops promoting pages. Only 46% of the traffics are accessed from the local node (Figure 19b). Throughput drops by 10%.

We can not setup AutoTiering with 1:4 configuration. It frequently crashes right after the warm up phase, when query fires. We run Cache1 with AutoTiering on 2:1 configuration. TPP out performs AutoTiering even with a 46% smaller local node – TPP can serve 10% more traffic from local node and provides 7% better throughput over AutoTiering.

6.3.2 Comparison between TPP and TMO

. TPP vs. TMO. TMO monitors application stalls during execution time because of insufficient system resources (CPU, memory, and IO). Based on the pressure stall information (PSI), it decides on the amount of memory that needs to be offloaded to the swap space. As mentioned in §4, using TMO for CXL-Memory's (z)swap-based abstraction is beyond our design goal. For the sake of argument, if we configure CXL-Memory as a swap-space for TMO, it will be only populated during reclamation. New page allocation can never happen there. Besides, without any fast promotion mechanism, aggressive reclamation can hurt application's performance; especially, when reclaimed pages are re-accessed through costly swap-ins. As a result, TMO throttles and can not populate most of the CXL-Memory capacity. For Web1, Cache1, and Data Warehouse in 2:1 configuration, TMO can only consume 45%, 61%, and 7% of the CXL-Memory capacity, respectively. On the other hand, TPP can use CXL-Memory for both allocation and reclamation purposes. For same applications, TPP's CXL-Memory usage is 83%, 92%, and 87%, respectively.

TPP with TMO. We run TMO with TPP and observe they are orthogonal and augment each other's behavior for Web1 on 2:1 configuration. TPP keeps most hot pages in local node; it gets slightly better when TMO is enabled (Table 3). TMO creates more free memory space in the system by swapping out cold pages both from local and CXL-Memory nodes. The presence of some memory headroom in the system makes it easier for TPP to move pages around and leads to fewer page migration failures. As TPP-driven migration fails less frequently, page placement is more optimized, resulting in even fewer accesses to the CXL-node.

TMO is still able to save memory without noticeable performance overhead, as shown in Table 4. This is because TPP makes (z)swap a two-stage process – TMO-driven reclamation in local node will first demote victim pages to CXL-Memory before getting (z)swapped out eventually. This improves victim page selection process – semi-hot pages now get a second chance for staying in local memory when drifting down CXL-node's LRU list. As a result, TPP reduces the amount of process stall in TMO originated from major page faults (i.e., memory and IO pressure in [73]) by 30%. As TMO throttles itself based on the process stall metric, this improves memory saving by 3% (2GB in absolute terms).

7 DISCUSSION AND FUTURE RESEARCH

TPP makes us production-ready to onboard our first generation of CXL-enabled tiered-memory system. We, however, foresee research opportunities with technology evolution.

Tiered Memory for Multi-tenant Clouds. In a typical cloud, when multiple tenants co-exist on a single host machine, TPP can effectively enable them to competitively share different memory tiers. When local memory size dominates the total memory capacity of the system, this may not cause much problem. However, if

applications with different priorities have different QoS requirements, TPP may provide sub-optimal performance. Integrating a well-designed QoS-aware memory management mechanism over TPP can address this problem.

Allocation Policy for Memory Bandwidth Expansion. For memory bandwidth-bound applications, CPU to DRAM memory bandwidth often becomes the bottleneck. CXL's additional memory bandwidth can help by spreading memory across the top-tier and remote node. Instead of only placing cold pages into CXL-Memory, which draw very low bandwidth consumption, the optimal solution should place the right amount of bandwidth-heavy, latency-insensitive pages to CXL-Memory. The methodology to identify the ideal fraction of such working sets may even require hardware support. We want to explore transparent memory management for memory bandwidth-expansion use case in our future work.

Hardware Support for Effective Page Placement. Hardware features can further enhance performance of TPP. A memory-side cache and its associated prefetcher on the CXL ASIC might help reduce the effective latency of CXL-Memory. Hardware support for data movement between memory tiers can help reduce page migration overheads. While in our environment we do not see a high migration overheads, others may chose to put provision systems more aggressively with very small amount of local memory and high amount of CXL-Memory. For our use cases, in steady state, the migration bandwidth is 4–16 MB/s (1–4K pages/second) which is far lower than CXL link bandwidth and also unlikely to cause any meaningful CPU overhead due to page movement.

8 RELATED WORK

Tiered Memory System. With the emergence of low-latency non-DDR technologies, heterogeneous memory systems are becoming popular. There have been significant efforts in using NVM to extend main memory [5, 31, 37, 38, 45, 46, 58, 60, 63]. CXL enables an intermediate memory tier with DRAM-like low-latency in the hierarchy and brings a paradigm shift in flexible and performant server design. Industry leaders are embracing CXL-enabled tiered memory system in their next-generation datacenters [1, 4, 9, 10, 21, 24, 25].

Page Placement for Tiered Memory. Prior work explored hardware-assisted [57, 59, 62] and application-guided [20, 25, 38, 72] page placement for tiered memory systems, which may not often scale to datacenter use cases as they require hardware support or application redesign from the ground up.

Application-transparent page placement approaches often profile an application's physical [29, 31, 36, 45, 48, 78] or virtual address-space [53, 63, 75, 77] to detect page temperature. This causes high performance-overhead because of frequent invocation of TLB invalidations or interrupts. We find existing in-kernel LRU-based page temperature detection is good enough for CXL-Memory. Prior study also explored machine learning directed decisions [36, 48], user-space APIs [53, 63], and swapping [31, 48] to move pages across the hierarchy, which are either resource or latency intensive.

In-memory swapping [8, 13, 26, 30] can be used to swap-out cold pages to CXL-node. In such cases, CXL-node access requires page-fault and swapped-out pages are immediately brought back to main memory when accessed. This makes in-memory swapping ineffective for workloads that access pages at varied frequencies.

When CXL-Memory is a part of the main memory, less frequently accessed pages can be on CXL-node without any page-fault overhead upon access.

Solutions considering NVM to be the slow memory tier [28, 43, 47, 76] are conceptually close to our work. Nimble [76] is optimized for huge page migrations. During migration, it employs page exchange between memory tiers. This worsens the performance as a demotion needs to wait for a promotion in the critical path. Similar to TPP, AutoTiereing [47] and work from Huang et al. [28] use background migration for demotion and optimized NUMA balancing [22] for promotion. However, their timer-based hot page detection causes computation overhead and is often inefficient, especially when pages are infrequently accessed. Besides, none of them consider decoupling allocation and reclamation paths. Our evaluation shows, this is critical for memory-bound applications to maintain their performance under memory pressure.

Disaggregated Memory. Memory disaggregation exposes capacity available in remote hosts as a pool of memory shared among many machines. Most recent memory disaggregation efforts [32–34, 40, 42, 51, 55, 56, 64, 66, 71] are specifically designed for RDMA over InfiniBand or Ethernet networks where latency characteristics are orders-of-magnitude higher than CXL-Memory. Memory managements of these systems are orthogonal to TPP– one can use both CXL- and network-enabled memory tiers and apply TPP and memory disaggregation solutions to manage memory on the respective tiers.

9 CONCLUSION

We analyze datacenter applications' memory usage behavior using Chameleon, a lightweight and robust user-space working set characterization tool, to find the scope of CXL-enabled tiered-memory system. To realize such a system, we design TPP, an OS-level transparent page placement mechanism that works without any prior knowledge on applications' memory access behavior. We evaluate TPP using diverse production workloads and find TPP improves application's performance on default Linux by 18%. TPP also outperforms NUMA Balancing and AutoTiering, two state-of-the-art tiered-memory management mechanisms, by 5–17%.

REFERENCES

[1] A Milestone in Moving Data. https://newsroom.intel.com/editorials/milestone-moving-data/.
[2] Alibaba Cluster Trace 2018. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md.
[3] AMD Infinity Architecture. https://www.amd.com/en/technologies/infinity-architecture.
[4] AMD Joins Consortia to Advance CXL. https://community.amd.com/t5/amd-business-blog/amd-joins-consortia-to-advance-cxl-a-new-high-speed-interconnect/ba-p/418202.
[5] Baidu feed stream services restructures its in-memory database with intel optane technology. https://www.intel.com/content/www/us/en/customer-spotlight/stories/baidu-feed-stream-case-study.html.
[6] CCIX. https://www.ccixconsortium.com/.
[7] Compute Express Link (CXL). https://www.computeexpresslink.org/.

[8] Creating in-memory RAM disks. https://cloud.google.com/compute/docs/disks/mount-ram-disks.

[9] CXL and the Tiered-Memory Future of Servers. https://www.lenovoxperience.com/newsDetail/283yi044hzgcdv7snkrmmx9ovpq6aesmy9u9k7ai2648j7or.

[10] CXL Roadmap Opens Up the Memory Hierarchy. https://www.nextplatform.com/2021/09/07/the-cxl-roadmap-opens-up-the-memory-hierarchy/.

[11] DAMON: Data Access MONitoring Framework for Fun and Memory Management Optimizations. https://www.linuxplumbersconf.org/event/7/contributions/659/attachments/503/1195/damon_ksummit_2020.pdf.

[12] Facebook and Amazon are causing a memory shortage. https://www.networkworld.com/article/3247775/facebook-and-amazon-are-causing-a-memory-shortage.html.

[13] Frontswap. https://www.kernel.org/doc/html/latest/vm/frontswap.html.

[14] Gen-Z. https://genzconsortium.org/.

[15] Google Cluster Trace 2019. https://github.com/google/cluster-data/blob/master/ClusterData2019.md.

[16] Idle Memory Tracking. https://www.kernel.org/doc/Documentation/vm/idle_page_tracking.txt.

[17] Idle page tracking-based working set estimation. https://lwn.net/Articles/460762/.

[18] Intel® Xeon® Processor Scalable Family Technical Overview. https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[19] Introducing new product innovations for SAP HANA, Expanded AI collaboration with SAP and more. https://azure.microsoft.com/en-us/blog/introducing-new-product-innovations-for-sap-hana-expanded-ai-collaboration-with-sap-and-more/.

[20] Memkind. https://memkind.github.io/memkind/.

[21] Micron Exits 3DXPoint, Eyes CXL Opportunities. https://www.eetimes.com/micron-exits-3d-xpoint-market-eyes-cxl-opportunities.

[22] NUMA Balancing (AutoNUMA). https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf.

[23] OpenCAPI. https://opencapi.org/.

[24] Reimagining Memory Expansion for Single Socket Servers with CXL. https://www.computeexpresslink.org/post/cxl-consortium-upcoming-industry-events.

[25] Samsung Unveils Industry-First Memory Module Incorporating New CXL Interconnect Standard. https://news.samsung.com/global/samsung-unveils-industry-first-memory-module-incorporating-new-cxl-interconnect-standard.

[26] The zswap compressed swap cache. https://lwn.net/Articles/537422/.

[27] Tmpfs. https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html.

[28] Top-tier memory management. https://lwn.net/Articles/857133/.

[29] Using DAMON for proactive reclaim. https://lwn.net/Articles/863753/.

[30] zram. https://www.kernel.org/doc/Documentation/blockdev/zram.txt.

[31] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. SIGPLAN, 2017.

[32] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In USENIX ATC, 2018.

[33] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker. Can far memory improve job throughput? In EuroSys, 2020.

[34] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli. Rethinking software runtimes for disaggregated memory. In ASPLOS, 2021.

[35] Y. Chen, I. B. Peng, Z. Peng, X. Liu, and B. Ren. ATMem: Adaptive data placement in graph applications on heterogeneous memories. In CGO, 2020.

[36] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In HPDC, 2019.

[37] J. Du and Y. Li. Elastify cloud-native spark application with PMEM. Persistent Memory Summit, 2019.

[38] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In EuroSys, 2016.

[39] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM footprint with NVM in Facebook. In EuroSys, 2018.

[40] Y. Gao, Q. Li, L. Tang, Y. Xi, P. Zhang, W. Peng, B. Li, Y. Wu, S. Liu, L. Yan, F. Feng, Y. Zhuang, F. Liu, P. Liu, X. Liu, Z. Wu, J. Wu, Z. Cao, C. Tian, J. Wu, J. Zhu, H. Wang, D. Cai, and J. Wu. When cloud storage meets RDMA. In NSDI, 2021.

[41] D. Gouk, S. Lee, M. Kwon, and M. Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In USENIX ATC, 2022.

[42] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In NSDI, 2017.

[43] D. Hansen. Migrate pages in lieu of discard. https://lwn.net/Articles/860215/.

[44] B. Holden, D. Anderson, J. Trodden, and M. Daves. HyperTransport 3.1 Interconnect Technology. 2008.

[45] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In ISCA, 2017.

[46] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. Dreslinski. Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension. In USENIX ATC, 2021.

[47] J. Kim, W. Choe, and J. Ahn. Exploring the design space of page management for Multi-Tiered memory systems. In USENIX ATC, 2021.

[48] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan. Software-defined far memory in warehouse-scale computers. In ASPLOS, 2019.

[49] S.-H. Lee. Technology scaling challenges and opportunities of memory devices. In 2016 IEEE International Electron Devices Meeting (IEDM), 2016.

[50] Y. Lee, Y. Kim, and H. Y. Yeom. Lightweight memory tracing for hot data identification. Cluster Computing, 2020.

[51] Y. Lee, H. A. Maruf, M. Chowdhury, A. Cidon, and K. G. Shin. Hydra : Resilient and highly available remote memory. In FAST, 2022.

[52] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In ASPLOS, 2023.

[53] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu. Utility-based hybrid memory management. In CLUSTER, 2017.

[54] C. A. Mack. Fifty years of moore's law. IEEE Transactions on Semiconductor Manufacturing, 2011.

[55] H. A. Maruf and M. Chowdhury. Effectively Prefetching Remote Memory with Leap. In USENIX ATC, 2020.

[56] H. A. Maruf, Y. Zhong, H. Wong, M. Chowdhury, A. Cidon, and C. Waldspurger. Memtrade: A disaggregated-memory marketplace for public clouds. arXiv preprint arXiv:2108.06893, 2021.

[57] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In HPCA, 2015.

[58] V. Mishra, J. L. Benjamin, and G. Zervas. MONet: heterogeneous memory over optical network for large-scale data center resource disaggregation. Journal of Optical Communications and Networking, 2021.

[59] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for nvm+dram hybrid main memory. In HotOS, 2009.

[60] M. Oskin and G. H. Loh. A software-managed approach to die-stacked dram. In PACT, 2015.

[61] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. SIGOPS Oper. Syst. Rev., 2010.

[62] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In ICS, 2011.

[63] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter. HeMem: Scalable tiered memory management for big data applications and real nvm. In SOSP, 2021.

[64] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In OSDI, 2020.

[65] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H.-C. Hoppe, and J. Labarta. Automating the application data placement in hybrid memory systems. In IEEE International Conference on Cluster Computing (CLUSTER), 2017.

[66] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In OSDI, 2018.

[67] A. Sriraman and A. Dhanotia. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. 2020.

[68] A. Sriraman, A. Dhanotia, and T. F. Wenisch. SoftSKU: Optimizing server architectures for microservice diversity @scale. In ISCA, 2019.

[69] Vladimir Davydov. Idle Memory Tracking. https://lwn.net/Articles/639341/.

[70] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In NSDI, 2020.

[71] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Netravali, M. Kim, and G. H. Xu. Semeru: A Memory-Disaggregated managed runtime. In OSDI, 2020.

[72] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen. Exploiting program semantics to place data in hybrid memory. In PACT, 2015.

[73] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos. TMO: Transparent memory offloading in datacenters. In ASPLOS, 2022.

[74] K. Wu, Y. Huang, and D. Li. Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory. In SC, 2017.

[75] K. Wu, J. Ren, and D. Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In SC, 2018.

[76] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. Nimble page management for tiered memory systems. In ASPLOS, 2019.

[77] L. Zhang, R. Karimi, I. Ahmad, and Y. Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. In SIGMETRICS, 2020.

[78] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In ASPLOS, 2004.