

TPP: Transparent Page Placement for CXL-Enabled Tiered Memory

Hasan Al Maruf^{*}, Hao Wang[†], Abhishek Dhanotia[†], Johannes Weiner[†], Niket Agarwal[†], Pallab Bhattacharya[†], Chris Petersen[†], Mosharaf Chowdhury^{*}, Shobhit Kanaujia[†], Prakash Chauhan[†]

University of Michigan^{*} Meta Inc.[†]

Abstract

With increasing memory demands for datacenter applications and the emergence of coherent interfaces like CXL that enable main memory expansion, we are about to observe a wide adoption of tiered-memory subsystems in hyperscalers. In such systems, main memory can constitute different memory technologies with varied performance characteristics.

In this paper, we characterize the memory usage of a wide range of datacenter applications across the server fleet of a hyperscaler (Meta) to get insights into an application’s memory access patterns and performance on a tiered memory system. Our characterizations show that datacenter applications can benefit from tiered memory systems as there exist opportunities for offloading colder pages to slower memory tiers. Without efficient memory management, however, such systems can significantly degrade performance.

We propose a novel OS-level application-transparent page placement mechanism (TPP) for efficient memory management. TPP employs a lightweight mechanism to identify and place hot and cold pages to appropriate memory tiers. It enables page allocation to work independently from page reclamation logic that is, otherwise, tightly coupled in today’s Linux kernel. As a result, the local memory tier has memory headroom for new allocations. At the same time, TPP can promptly promote performance-critical hot pages trapped in the slow memory tiers to the fast tier node. Both promotion and demotion mechanisms work transparently without any prior knowledge of an application’s memory access behavior.

We evaluate TPP with diverse workloads that consume significant portions of DRAM on Meta’s server fleet and are sensitive to memory subsystem performance. TPP’s efficient page placement improves Linux’s performance by up to 18%. TPP outperforms NUMA balancing and AutoTiering, state-of-the-art solutions for tiered memory, by 10–17%.

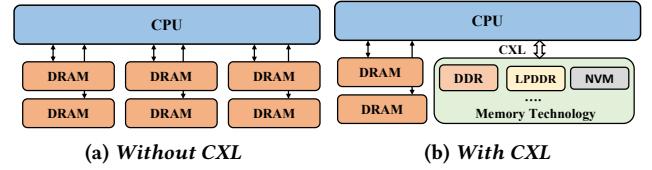


Figure 1: CXL decouples memory from compute.

1 Introduction

The surge in memory needs for datacenter applications [12, 56], combined with the increasing DRAM cost and technology scaling challenges [46, 49], 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, 18, 36, 37, 43]. These technologies, however, have much higher latencies vs. main memory and can significantly degrade performance when data is inefficiently placed in different levels of the memory hierarchy. Prior knowledge of application behavior and careful application tuning can address this, which is 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. By decoupling the memory device choice from the CPU, CXL allows more flexibility in memory subsystem design and fine-grained control over the memory bandwidth and capacity [9, 10, 23]. Additionally, as CXL-Memory appears like the main memory, it provides opportunities for application-agnostic transparent page placement on the appropriate memory tier.

To leverage the benefits of memory tiering and build efficient page placement techniques, we first need to understand the variety of memory access behavior in 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, 16, 63] do not fit the bill as they require kernel modification, which is often not possible in production machines. To this end, we build Chameleon, a robust user-space tool, that uses existing CPUs’ Precise Event-Based Sampling (PEBS) mechanism to characterize an application’s memory access behavior (§3). We sample the LLC and TLB misses to get information on the most recent load and store operations in the virtual address space and generate a heatmap for an application’s usage on different types of memory pages.

We use Chameleon to profile a variety of large memory-bound applications across different service domains running in Meta’s production environment and make the following observations. **(1)** Workloads have meaningful portions of warm/cold memory. 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 provides enough opportunity to observe application behavior and make page placement decisions in kernel-space. **(4)** Workloads have different sensitivity levels toward anon and file pages.

In a tiered memory system, average memory access latency varies across memory tiers (Figure 2) and application performance greatly depends on the fraction of memory served from the fast memory. Linux kernel’s memory management mechanism is designed for homogeneous CPU-attached DRAM-only systems and performs poorly on CXL-Memory system. Its paging-based reclamation mechanism for moving cold pages to slow memory tiers is extremely inefficient for a 100 nanosecond-granular CXL-Memory. Besides, when fast memory tier is under pressure, new allocations often move to slow memory tier if the reclamation mechanism fails to cope up with the allocation rate. As a result, hot pages often get trapped in the slow tier and experience high access latency.

Considering the above observation, we design an OS-level transparent page placement mechanisms (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 light-weight reclamation mechanism to demote colder pages to the slow tier node; **(b)** decoupling the

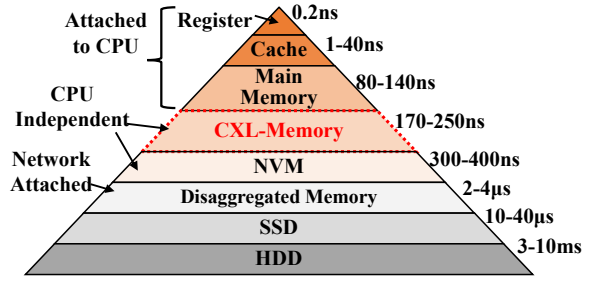


Figure 2: Different tiers has different characteristics with varied memory access latency in a heterogeneous tiered-memory system.

allocation and reclamation logic 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 that can benefit an application upon being promoted to the fast memory tier. We also introduce support for page type-aware allocation across the memory tiers, e.g., preferably allocate anon pages to fast tier and file caches to slow tier. With this optional application-aware preference 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 Meta fleet and run them on a CXL-enabled system (§6). We find that TPP provides the similar performance behavior of all memory served from the fast memory tier. For Cache workloads, this holds even when DRAM can host only 20% of the working set. 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 [21] and AutoTiering [44], two state-of-the-art page placement mechanisms for tiered memory. TPP outperforms both of them by 10–17%.

We make the following contributions in this paper:

- We present Chameleon, a lightweight user-space memory characterization tool. We use it to understand production workload’s memory consumption behavior and assess the scope of tiered-memory in hyperscale datacenters (§3). We plan to open source Chameleon.
- We propose TPP for efficient memory management on a tiered-memory system (§5). We have published the source code of TPP for a Linux upstream discussion.
- We evaluate TPP on CXL-enabled tiered memory systems with real-time production workloads (§6). For datacenter applications, TPP improves default Linux’s performance by up to 18%. TPP also outperforms NUMA Balancing and AutoTiering by 10–17%.

We are the first to characterize and evaluate an end-to-end practical system using CXL-Memory 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 leads to a rapid growth in memory demands across the Meta server fleet. With new generations of CPU and DRAM technologies, memory is becoming the more prominent source of expenses in the rack-level total cost of ownership (TCO). With every subsequent server generation, memory cost and power constitute an increasingly higher % of overall rack TCO (Figure 3).

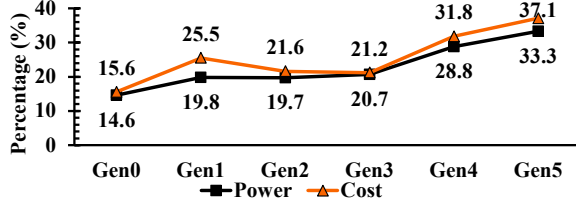


Figure 3: Memory as a percentage of rack TCO and power across different hardware generations.

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) which 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, 17, 41] and not commonly supported across all the CPUs [6, 14, 22]. 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

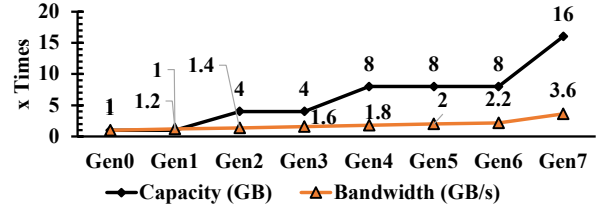


Figure 4: Memory BW and capacity don’t increase proportionately.

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 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, 20, 23, 24]. All the tools, drives, 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 of the time [2, 15, 45, 64]. Often an application allocates a large amount of memory but accesses it infrequently [45, 51]. We characterize four popular applications in Meta’s production server fleet and find that 55-80% of an application’s allocated memory remains idle within any two minutes interval (§3.3). 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.

3 Memory Usage of Datacenter Applications

To understand the scope of CXL-based tiered memory system for datacenter applications, we characterized several memory-intensive applications in our production fleet. The goal of this characterization is to answer a few high-level

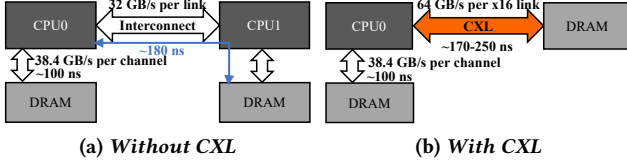


Figure 5: CXL-Memory has similar bandwidth with NUMA-like latency characteristics of a dual-socket server.

questions, including if/how a tiered memory solution can help, and what kind of page placement approach would work across a variety of applications that run in datacenters.

Existing memory characterization techniques fall short of providing these insights. For example, the commonly used IPT-based characterization [11, 16, 63] requires kernel support for exposing a page’s idle access bit to the user space. To enable this, one needs to change production machine’s kernel, which is not scalable. Moreover, IPT only provides information in the physical address space – we cannot track memory allocation/deallocation if a physical page is re-used by multiple virtual pages. In this work, we propose Chameleon, a light-weight user-space memory characterization tool to understand a workload’s memory access behavior.

3.1 Overview of Chameleon

Chameleon consists of two primary components – a Collector and a Worker – running simultaneously on two different threads. The Collector utilizes the PEBS mechanism of modern processor’s performance monitoring unit (PMU) to collect hardware-level performance events related to memory accesses. The Worker uses the sampled information to generate insights on a workload’s memory access behavior.

Collector In modern CPUs, when the total number of a particular hardware event exceeds a specified threshold, PMU generates an overflow interrupt and dumps the hardware event records to a software-configured sampling buffer. To capture the memory access information, the Collector samples last level cache misses for demand loads (event `MEM_LOAD_RETIRED.L3_MISS`) and TLB misses for demand stores (event `MEM_INST_RETIRED.ALL_STORES`). The sampled records provide us with the PID and virtual memory address target for the memory access events.

Like any other sampling mechanism, the accuracy of PEBS depends on the sampling rate – frequent samples will provide higher accuracy. High sampling rate, however, demands larger sampling buffer and more CPU resources. In our fleets, we find that one sample for every 200 events is a good trade-off point for balancing between overhead and accuracy.

To avoid CPU stalling, the Collector divides all CPU cores into a set of groups and enables sampling on one group 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 tune the trade-off between overhead and accuracy. The Collector reads the sampling buffer and writes into one of a two hash table set. After each interval (by default, 1 minute), the Collector wakes up the Worker to process data on 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. It capitalizes kernel provided user-space-level per-process memory mappings (`/proc/$PID/maps` and `/proc/$PID/pagemap`) to collect information on a page’s physical address, memory type, and locality. This address translation can cause high overhead if the target process’ working set size is extremely large (e.g., terabyte-scale). One can, however, configure the Worker to disable the physical address translation and characterize an application only on its virtual-space access pattern.

For each page, we maintain a 64-bit bitmap to track 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 also configure the Worker to use multiple bits for one interval to capture the difference in page access frequency, at the cost of supporting shorter history. After generating the statistics and reporting them, the Worker sleeps (Figure 6c).

A common concern for PEBS-based profiling is the performance regression due to interrupts. In Meta’s production environment, CPU utilization is rarely driven to 100% even at peak load. More importantly, 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. We do not see any noticeable application-level performance variance while running Chameleon in production environments.

3.2 Production Workload Overview

We use Chameleon to characterize memory-bound datacenter applications. We pick four most popular production applications running across Meta’s server fleet serving live traffic on three diverse service domains. These workloads constitute a significant portion of the server fleet and represent a wide variety of Meta workloads [61, 62].

Web. Web implements a Virtual Machine with Just-In-Time (JIT) compilation and runtime system to serve web

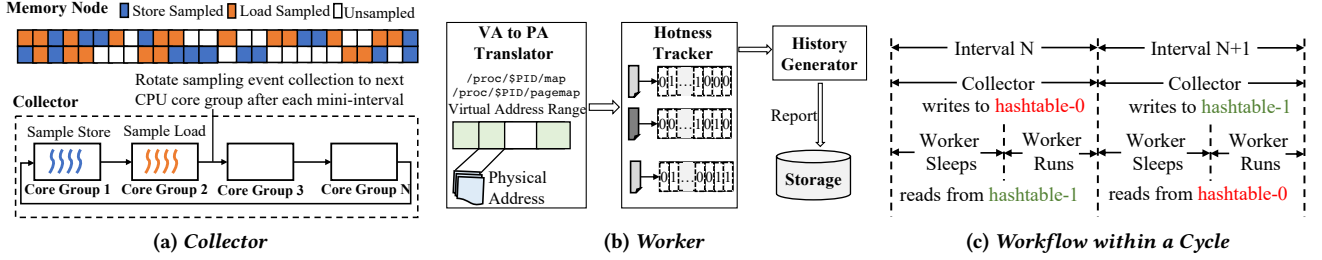


Figure 6: Chameleon runs Collector and Worker threads separately to collect and process memory access related HW events independently.

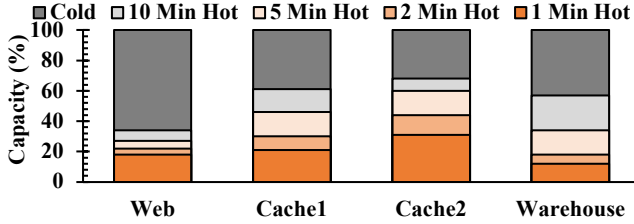


Figure 7: Workloads maintain significant amount of cold memory that can be offloaded to a slow memory tier.

requests originating from end-users. Web employs request-level parallelism – an incoming request is assigned to one of a fixed pool of worker threads, which serves the request until completion.

Cache. Cache is a large distributed-memory object caching service lying between the web and database tiers for low-latency data-retrieval. Cache1 and Cache2 correspond to two tiers within each geographic region for this service. Client services contact the Cache1 tier first. If misses, it is forwarded to the Cache2 tier. Cache2 misses are then sent to an underlying database cluster in that region.

Data Warehouse. 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. A query served by this service can consume terabytes of memory and may take days to complete. To maintain fault-tolerance, this service writes the output from each stage to disk.

3.3 Page Temperature

In datacenter applications, a significant amount of allocated memory remains cold beyond a few minutes of intervals (Figure 7). For example, Web uses 97% of the system’s total memory capacity, but within a two-minute interval, it uses only 22% of the total allocated memory on average. Same goes for the Cache applications. While the memory consumption is within 95%–98% of the system’s total capacity, only 30%–40% of them are accessed within a two-minute interval.

While Web and Cache applications mostly serve user requests, Data Warehouse is a compute-heavy workload where

a specific computation operation can span even terabytes of memory. As expected, this workload consumes almost all the available memory within a server. However, even for this workload, on average, only 20% of the accessed memory remains 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.

3.4 Page Temperature Across Different Page Types

Applications consume different types of memory 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% of the total allocated anons remain hot; for files, in contrast, it is only 14% of the total allocation (Figure 8).

Cache applications use tmpfs [26] for a fast in-memory lookup. Anons are used mostly for processing queries. As a result, file pages contributes significantly to the total hot memory. However, even for Cache1, 40% of the anons get accessed within every two minutes, while for file the fraction is only 25%. For Cache2, the fraction of anon and file usage is almost similar within a two-minute time window – 43% and 45% of their total allocation, respectively. This is because, more file pages get touched than anons during a cache lookup. However, within a minute interval, even for Cache2, higher fraction of anons (43%) remain hot over file pages (30%).

Data Warehouse uses 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 tend to be hot, while file pages remain cold within short intervals.

3.5 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

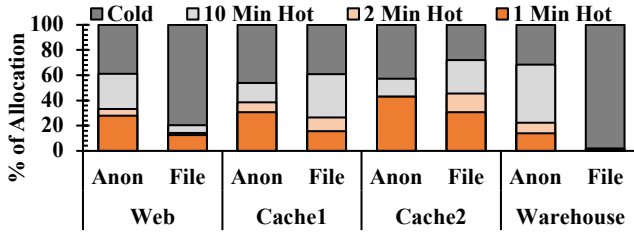


Figure 8: Fraction of anons tends to be hotter than the files.

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 look-ups. As a result, file pages consume most of the allocated memory. For Cache1 (Figure 9b) and Cache2 (Figure 9c), the fraction of file caches hover around 70–82% and 75–80%, respectively. While the fraction of anon and file remains almost steady, if at any point of time, anon usage grows, file caches are discarded to make space for the newly allocated anons.

For Data Warehouse workload, anon pages consume most of the allocated memory – 85% of 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.

3.6 Impact of Page Types on Application Throughput

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

For Cache applications, tmpfs is allocated during application 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 the high 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 memory types.

3.7 Page Re-access Time Granularity

Cold memories often get re-accessed at a later point of time. 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 minutes interval. This indicates Web mostly repurposes pages allocated at an earlier time. Same goes for the Cache applications. For these applications, discarding the cold memory can impact application-level performance as a good chunk of colder pages get re-accessed within any 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.

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

From the above observations, it is obvious that tiered memory subsystems can be a good fit for datacenter applications as there exists a significant portion of cold memory with steady access patterns. In the subsequent sections, we will first discuss the shortcomings of default Linux in a tiered memory system (§4) and how that motivates us to design a transparent page placement mechanism for such a system (§5).

4 Background on Linux Page Placement Policy

Page management policies of existing Linux kernels are designed assuming that a system will have homogeneous DRAM-only NUMA nodes. More importantly, having CPU-less memory nodes changes the locality assumptions (pages accessed by a CPU core needs to be in its local memory node) in existing NUMA memory managements. In this section, we discuss Linux’s shortcomings for tiered memory systems.

4.1 Page Allocation and Reclamation

By default, Linux kernel tries to allocate a page to the memory node local to a CPU where the process is running. To track whether a node has enough free pages for the allocation, it maintains three watermarks (min_watermark, low_watermark, high_watermark) for each memory zone within that 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 (❶ in Figure 12a). It initially tries to free up inactive file caches. If that is not enough, it then moves to free up anon and mapped file pages. At this point, the reclaimers initiates paging mechanism – it accesses swapping device to page out

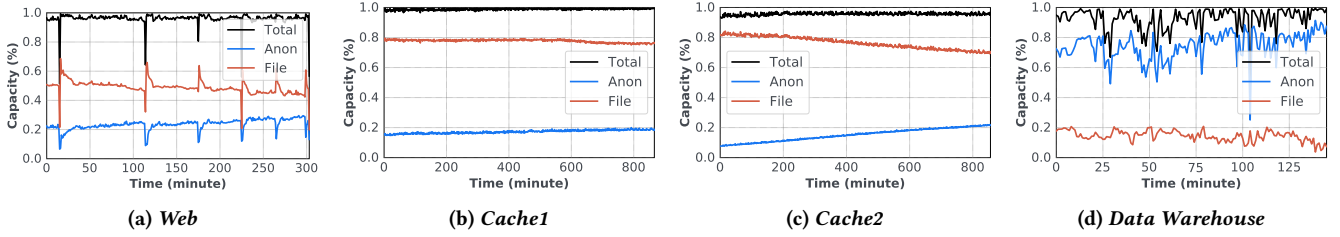


Figure 9: Different types of memory usage varies over time for different workloads.

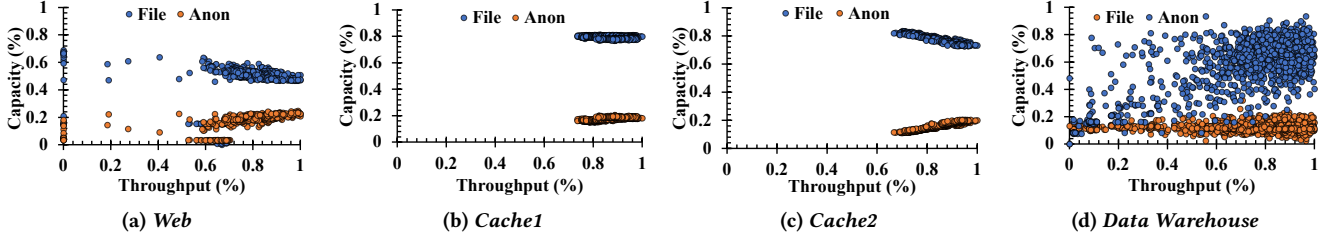


Figure 10: Workloads tend to provide higher throughput at higher memory utilization. They have varied sensitivity towards anon and file pages.

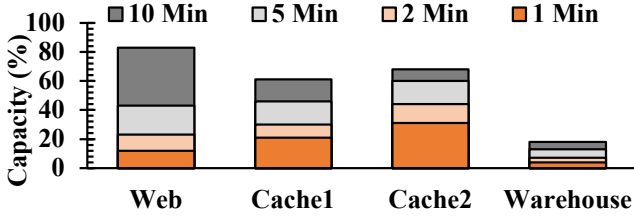


Figure 11: Fraction of pages re-accessed at different time interval

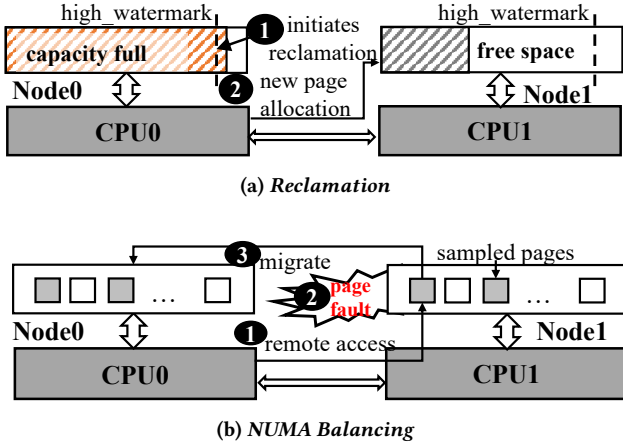


Figure 12: Reclamation and NUMA Balancing in default Linux.

the colder memory. Invoking paging events in the critical path worsens the average page access latency and impacts application performance. In a multiple NUMA node system, during this reclamation process, the page allocator checks next eligible memory nodes to grab free pages and new pages

get allocated to the remote CPU’s memory node (② in Figure 12a). Accessing the pages from the remote memory node causes extra NUMA latency as the remote CPU gets involved in the path.

When the reclaimer frees up enough memory to satisfy local memory node’s `high_watermark`, allocation can happen on the local node again. Both allocation and reclamation check the `high_watermark` to decide on when to allocate new pages and when to stop reclaiming, respectively. With high allocation rate, reclamation may fail to cope-up as it is slower than allocation. In such a case, memory retrieved by the reclaimer fills up soon to satisfy allocation requests. As a result, local memory allocations halt frequently and degrade application performance.

4.2 NUMA Balancing to Reduce Access Latency

If pages could be allocated to and accessed from a CPU’s local memory nodes, average memory access latencies would be minimum and application performance would be optimal. To minimize the access latency from a CPU’s remote memory node, one can use NUMA balancing (also known as AutoNUMA) [21], an existing Linux kernel feature, that moves pages to a memory node close to the CPU. A kernel task routinely samples a subset of each process’s allocated memory (by default, 256MB of pages) on each memory node and clears the present bit on their flags. When a CPU accesses a sampled page (① in Figure 12b), a minor page-fault is generated (known as NUMA hint fault). In the page-fault handler (② in Figure 12b), it records which CPU is trying to access the page. Pages that are accessed from a remote CPU are migrated to that CPU’s local memory node (③ in

Figure 12b) (known as promotion). NUMA balancing, however, assumes memory nodes are attached to CPUs, therefore, cannot move pages to a CPU-less node (e.g., CXL-Memory).

Before promotion, NUMA balancing checks whether the target node has enough free pages to support (i.e., `high_watermark` is satisfied). If the target node is under pressure, it avoids migration. At that point, the benefit of NUMA balancing disappears; pages remain trapped in the remote memory node and experience extra NUMA latency. Hot pages trapped to remote node often worsen the performance.

Takeaway. Linux uses paging mechanism to move cold pages. This is latency-intensive and not appropriate to CXL-Memory for main memory expansion. NUMA balancing uses migration for page movement. But, it cannot move pages to a CPU-less memory node. Besides, allocation and reclamation being tightly coupled, default Linux fails to maintain a headroom of free pages under memory pressure. This restricts both new allocation and promotion to local memory node.

5 Transparent Page Placement for CXL-Memory

Considering the pitfalls of Linux kernel’s existing page placement mechanism, we propose Transparent Page Placement (TPP) – a smart mechanism for tiered memory system. TPP is an OS-managed approach to place ‘hotter’ pages in local memory and move ‘colder’ pages in CXL-Memory.

TPP’s design-space can be divided across three main areas – lightweight demotion to CXL-Memory, decoupling allocation and reclamation path, and hot-page promotion to local nodes. TPP also introduces page type-aware allocation, e.g., prefer anon pages to local node while prefer cache pages to the CXL-node. In this section, we discuss all of the TPP components.

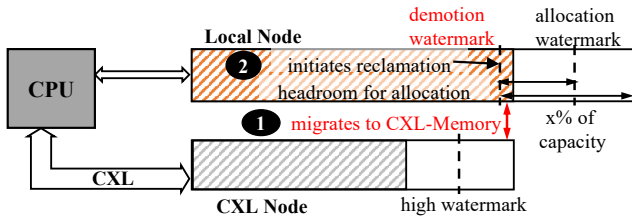


Figure 13: TPP decouples the allocation and reclamation logics for local memory node. TPP uses migration for page demotion.

5.1 Migration for Lightweight Reclamation

As mentioned in §4.1, when local memory node fills up, default Linux reclamation pages-out to swap device. 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.

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 (① in Figure 13). Migration to a NUMA node is orders of magnitude faster than accessing disks while swapping. We use the default LRU-based mechanism in existing Linux kernel to select the demotion candidates. However, unlike swapping, as demoted pages are still available in-memory, besides inactive file pages, we look for inactive anon pages while scanning reclamation candidates. 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 chose the demotion target based on a CXL-node’s state, this simple distance-based static mechanism turns out to be effective for a wide-range of production workloads.

5.2 Decoupling Allocation and Reclamation

When a local node is restricted by severe memory constraint, we need to proactively maintain some free memory headroom on the local node. This will help in two ways. First, new allocations (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` (② in Figure 13). Note that demotion watermark is always set to a higher value above the allocation watermark so that we always reclaim more to maintain the free memory headroom.

To control 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 process on local nodes. By default, its value is set to 2% that means reclamation starts as soon as only a 2% of the local node’s capacity is available to consume.

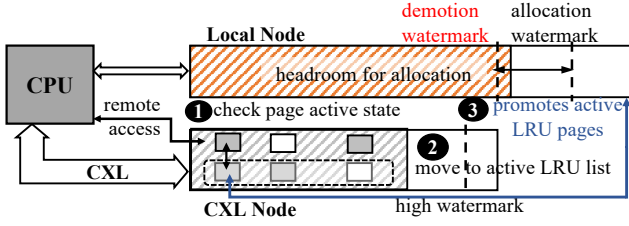


Figure 14: TPP uses page active state to select promotion candidate.

5.3 Page Promotion from Remote Nodes

Due to an increased memory pressure on local nodes, new pages may often be allocated to the CXL-nodes. Besides, demoted pages may also become hot globally. Without any promotion mechanism, this hot pages will always be trapped to the CXL-nodes and hurt the application performance. To promote a CXL-node’s pages, we augment Linux’s existing NUMA balancing mechanism. Maintaining the backward compatibility, we introduce a `NUMA_BALANCING_TIERED` mode to the existing `sysctl` user-space interface to enable NUMA balancing for tiered-memory systems. If a system starts with the default `NUMA_BALANCING` mode, but has only a single local node online, we automatically detect it and downgrade the system to `NUMA_BALANCING_TIERED` mode.

NUMA Balancing for CXL-Memory. In a tiered memory system, it is not reasonable to promote hot memories of a local node to other local or CXL-nodes. Sampling and poisoning pages to find a local node’s hot memory generate unnecessary NUMA hint fault overheads in a CXL-enabled tiered system. We avoid this by limiting sampling only to CXL-nodes.

As mentioned in §4.2, before initiating a promotion, NUMA balancing always checks whether the local node is under pressure and ignores the promotion request if the local memory is low on free pages. As a result, hot pages remain trapped in the CXL-nodes. Decoupling the allocation and reclamation watermark can solve this problem to a great extent. To create even more memory pressure to initiate the reclamation of colder pages on a local node, we ignore the allocation watermark checking while promoting a CXL-node’s hot pages. If a system has multiple local nodes, we select the node where the task is running. When applications share memory across the memory nodes, we chose the 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 being promoted, these type of pages may shortly become the demotion candidate if the local nodes are always under pressure. Thus, promotion traffics generated from infrequently accessed pages

can easily fill up the local node’s reclaimed free spaces and eventually generate a higher demotion traffic for CXL-nodes. This demotion-promotion ping-pong causes unnecessary traffic over the memory nodes and can negatively impact on the performance of memory-bound applications.

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 as the promotion candidate 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 (either of anon or file active LRU) (① in Figure 14). This significantly reduces the promotion traffic and always maintain a satisfactory amount of free memory on the local node to support both new allocations and promotions.

However, the main purpose of the OS maintained LRU lists is to have a relative classifications of pages 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 a CXL-node may not always be under pressure, most of the 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 so that the page gets moved in to the active LRU list immediately (② in Figure 14). 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 (③ in Figure 14).

Active LRU-based page promotion helps TPP add some hysteresis to page promotion and only promote pages that are atleast accessed twice within a reasonable time window. This, eventually, reduces unnecessary promotion traffic and speeds up the convergence of hot pages across the memory nodes.

5.4 Page Type-Aware Allocation

The page placement mechanism we described above is generic to all page types and can dynamically move pages around so that hottest pages eventually moves to the local node. However, some workloads have obvious memory access pattern and page type-aware allocation can help them even from the beginning. For example, production workloads often perform lots of file I/O during the warm up phase and generate file caches. Although these file caches consume a good portion of a system’s memory capacity, they are accessed very infrequently. As a result, cold file caches eventually end up demoted to the CXL-nodes. Not only that, local memory node being occupied by the inactive file caches

often forces anon memory 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 preferably, 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.

5.5 Observability into TPP to Assess Performance

Promotion- and demotion-related statistics can help better understand the effectiveness of the page placement mechanism. 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 (`pgdemote_anon`, `pgdemote_file`) 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. `pgpromote_candidate_demoted` tracks the number of demoted pages that become promotion candidates. High value of this counter means, TPP is thrashing across NUMA nodes.

Promotion can fail due to many reasons, e.g., local node having low memory, page reference count being abnormal, whole system being low on memory, etc. We add separate counters to track each of the promotion failure scenario to reason about where and how promotion fails.

6 Evaluation

We integrate TPP on Linux kernel v5.12. We evaluate TPP with production workloads mentioned in §3.2 serving live traffic on tiered memory systems across Meta’s server fleet. We explore the following questions:

- How effective TPP is in distributing pages across memory tiers and improving application performance? (§6.2)
- What are the impacts of different TPP components? (§6.3)
- How TPP performs against state-of-the-art page placement solutions? (§6.4)

6.1 Experimental Setup

We deploy a number of pre-production x86 CPUs that support CXL 1.1 specification. In those machines, we have FPGA-based CXL memory expansion card. Memory attached to that expansion card shows up to the OS as a CPU-less NUMA node. Although our current FPGA-based CXL cards has around 250ns higher latency than our eventual target, we use them for the functional validation in this work.

We have confirmation from two major x86 CPU vendors that the access latency to CXL-memory is similar as the remote latency on a dual-socket system. That’s why, for performance evaluation, we primarily use dual-socket systems and configure them to mimic our target CXL-enabled systems. We turn a typical dual-socket systems into a CXL-like systems with one CPU socket and one CPU-less memory node using following mechanisms:

- We effectively disable all CPU cores on socket-1 by excluding them from scheduler via `isolcpu` kernel option and binding all NIC channels to socket-0 CPU cores;
- We configure per-socket memory capacity and bandwidth via channel/rank-disabling BIOS option and `mmap` kernel option.
- We fine-tune the interconnect frequency on socket-1 to have the idle-state remote NUMA latency match the CXL-memory latency projection from our CXL ASIC vendor.

For baseline, we just disable all memory on socket-1 while enabling sufficient memory on socket-0. In the following sub-sections, we compare TPP against default Linux kernel (in §6.2), and default NUMA Balancing and AutoTiering, two state-of-the-art page placement mechanisms for tiered-memory system (in §6.4).

6.2 Effectiveness of TPP

To understand TPP’s benefit, we first consider a setup where local node can support all the hot portions of a working set of an application. We use the target configuration of Meta’s CXL-enabled systems in the production where the ratio of local node and CXL-Memory capacity is 2:1. In this setup, local node is supposed to serve all the hot traffic (§6.2.1).

To stress-test TPP on a constrained memory scenario, we configure the local node’s size such that it can support only a fraction of the total hot working set and pages are forced to move to the CXL-node. Here, the ratio of local and CXL-Memory capacity is 1:4 (i.e., 80% of the total working set resides on CXL-node) (§6.2.2). This simulates the case where memory capacity can be largely expanded through CXL.

6.2.1 Default Production Environment

Web. As mentioned in §3.2, Web performs lots of file I/O during initialization and fills up the local node. Due to the

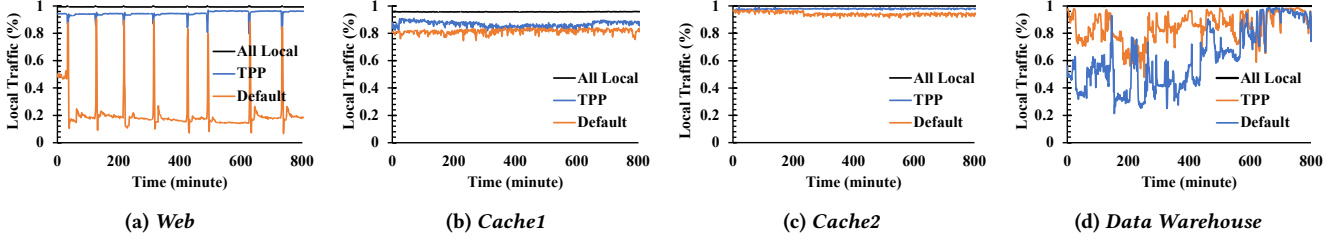


Figure 15: Fast demotion of cold pages and effective promotion of hot pages allow TPP to serve most of the traffics from the local node.

slow reclamation path of default Linux (44× slower reclamation rate than TPP), it takes much longer time to free up the local node. As a result, new allocation to the local node halts and anon pages get allocated to the CXL-node. Without any promotion mechanism, anon pages stay there forever. In default Linux kernel, on average, 78% of total memory accesses are served from the CXL-node (Figure 15a). As a result, the throughput drops by 16.5% over the all-from-local-setup.

In TPP, active and faster demotion helps move colder file pages to CXL-node and allow more anon pages to be allocated in the local node. With TPP, 92% of the anon pages are served from the local node, while it was only 55% with the default kernel. As a result, local node serves 90% of total memory accesses. Throughput improves over the default kernel and matches 99.5% of the all from local setup.

Cache. In §3.2, we observe Cache applications maintain a steady ratio of anon and file pages throughout their life-cycle. As a result, almost all the anon pages get allocated to the local node from the beginning and served from there.

For Cache1, although 30% of the total working set gets trapped in to the CXL-node, read traffic from CXL-node is only 8%. As a result, application performance remains very close to the all from local setup – performance regression is only 3%. TPP can minimize this performance gap (throughput is 99.9% of the all from local setup) by promoting all the trapped hot files back to the local node and improving the fraction of traffic served from the local node (Figure 15b).

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 10% of the total anon memory to the CXL-node. This allows the promotion of more hot file pages – 9% of the total file pages moves back to the local node. On a default Linux kernel, local node serves 78% of the memory accesses (Figure 15c). Average memory access latency increases by 14% with a 2% throughput regression. TPP improves the fraction of local traffic to 91%. Throughput regression is only 0.4% over the all from local setup.

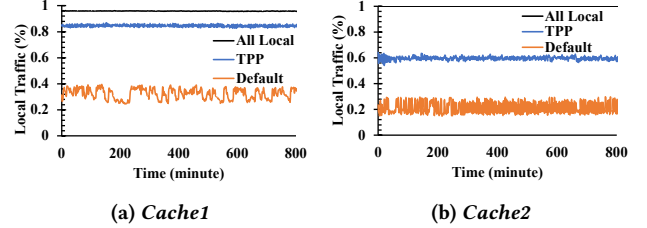


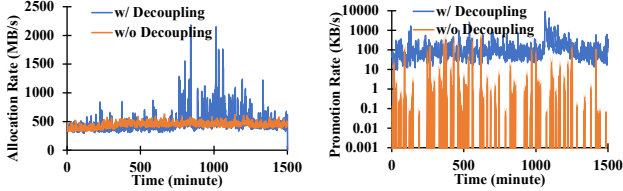
Figure 16: Effectiveness of TPP under memory constraint.

Data Warehouse. Data warehouse generates file caches to store the intermediate processing data. Most of the time, file caches remain cold. Besides, only one third of the total anon pages remain hot. As a result, the 2:1 memory configuration is good enough to serve all the hot memory from the local node. Both default Linux and TPP kernels perform similar as the all from local setup (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% of the anon pages. To make space for the hot anon pages, cold file pages are demoted to the CXL-node. Eventually, 92% of the total memory accesses happen from the local node. For default Linux kernel, local node can serve 88% of the total traffic (Figure 15d).

6.2.2 Large Memory Expansion with CXL In this section, we will observe TPP’s effectiveness during stress situations where hot memory is bound to spill in to the CXL-node. Extreme memory setups like 1:4 configuration is useful in flexible server design with DRAM as a small-sized local node and CXL-Memory as a large but cheaper memory. As in production, such a configuration is impractical for Web and Data Warehouse applications, we limit our discussion to the Cache applications only. Note that TPP is effective even for Web and Data Warehouse in such a setup and provides performance very close to the all from local setup.

Cache1. In a 1:4 configuration, on a default Linux kernel, 85% of the total anon pages get trapped to the CXL-node. File pages consume almost 90% of the local node’s capacity. As anon pages are most frequently accessed pages, almost 75% of the



(a) New Allocation (b) Promotion to Toplevel Node
Figure 17: Impact of decoupling allocation and reclamation.

total memory accesses happens from CXL-node (Figure 16a). Throughput drops by 14% drop over all-local setup.

As the local node is very small, with TPP, anon pages often get allocated to the CXL-node. However, because of the apt promotion mechanism, TPP can eventually promote almost all the remote hot anon pages (97% of the total hot anon pages within a minute interval) to the local node. Promotion of hot anon pages forces less latency-sensitive file pages to be demoted to the CXL-node. 87% of the total hot file pages within a minute interval are served from the CXL-node. Eventually, TPP stabilizes the traffic between the two nodes and 85% of the total reads are served from the local node. This helps Cache1 to reach the performance of all-local setup – even though the local node’s capacity is only 20% of the working set size, the throughput regression is only 0.5%.

Cache2. Similar to Cache1, on a default kernel, Cache2 experiences 18% throughput loss. Only 14% of the anon pages remain on the local node; whereas in 2:1 setup, 99% of the anon pages are served from the local node. As a result, CXL-node traffic increases to 80%.

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 file caches are now mostly in CXL-node, 41% of the read traffic comes from the CXL-node. Yet, the performance regression is only 5% over the all-local setup.

6.3 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. Decoupling allocation and reclamation helps TPP maintain a good headroom for new allocations and promotions to take place on the local node. 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, newly allocated pages are often short-lived (less than a minute life-time) and de-allocated even before being selected as a promotion candidate. Hot pages trapped to CXL-node worsens the performance.

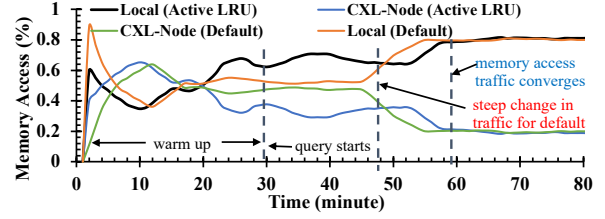


Figure 18: Restricting the promotion candidate based on their age reduces unnecessary promotion traffic.

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

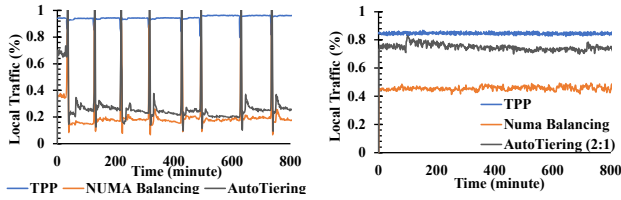
Table 1: Page-type aware allocation helps certain applications.

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 \times at the 95th percentile.

As the local node is always under memory pressure, new allocations consume the freed up pages instantly and promotion fails as the target node becomes low on memory after serving the 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 reduces CXL-node traffic to 15% and helps TPP to maintain the throughput of all-local setup.

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 \times . 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 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 applications, preferring the file cache allocation



(a) Web 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 AutoTiering on 2:1.

to CXL-node can provide all-local performances even with a small-sized local node (Table 1). TPP is efficient enough to keep most of the effective hot pages on the local node. Performance regression over the baseline is only 0.2–2.5%.

6.4 Comparison Against Existing Solutions

We compare TPP against Linux’s default NUMA Balancing and AutoTiering. We use Web 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 (§6.2).

Web. NUMA Balancing can help an application by promoting remote pages when the reclaim mechanism can provide with enough free pages on local node. However, when the local node is low on memory, NUMA Balancing stops promoting pages and performs even worse than a default Linux kernel because of its extra scanning and failed promotion tries. Due to the slow reclamation rate (42× slower than TPP), local node mostly remains occupied and promotion rate becomes very low (on average, 11× slower than TPP). 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 of local pages, NUMA Balancing’s system-wide CPU overhead is 2% higher over 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%.

As mentioned in (§6.2), TPP experiences only a 0.5% throughput regression over all-local setup.

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 46% smaller local node, TPP can serve 10% more traffic from local. TPP provides 7% better throughput over AutoTiering.

7 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, 30, 35, 36, 42, 43, 53, 55, 58]. 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, 20, 23, 24].

Page Placement for Tiered Memory. Prior work explored hardware-assisted [52, 54, 57] and application-guided [19, 24, 36, 66] 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 [28, 30, 34, 42, 45, 70] or virtual address-space [48, 58, 67, 69] 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 [34, 45], user-space APIs [48, 58], and swapping [30, 45] to move pages across the hierarchy, which are either resource or latency intensive.

In-memory swapping [8, 13, 25, 29] 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 [27, 40, 44, 68] are conceptually close to our work. Nimble [68] 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, AutoTiering [44] and work from Huang et al. [27] use background migration for demotion and optimized NUMA balancing [21] 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 [31–33, 38, 39, 47, 50, 51, 59, 60, 65] 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.

8 Conclusion

We analyze datacenter applications’ memory usage behavior using Chameleon, a robust user-space working set characterization tool, to find the scope of CXL-enabled tiered memory system in production. To realize such a system, we design TPP, an OS-level efficient page placement mechanism that works transparently without any prior knowledge on applications’ memory access behavior. We evaluate TPP using diverse production workloads and find TPP can improve 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 10–17%.

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 page tracking-based working set estimation. <https://lwn.net/Articles/460762/>.
- [17] 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>.
- [18] 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/>.
- [19] Memkind. <https://memkind.github.io/memkind/>.
- [20] Micron Exits 3DXPpoint, Eyes CXL Opportunities. <https://www.eetimes.com/micron-exits-3d-xpoint-market-eyes-cxl-opportunities>.
- [21] NUMA Balancing (AutoNUMA). https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf.
- [22] OpenCAPI. <https://opencapi.org/>.
- [23] Reimagining Memory Expansion for Single Socket Servers with CXL. <https://www.computeexpresslink.org/post/cxl-consortium-upcoming-industry-events>.
- [24] 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>.
- [25] The zswap compressed swap cache. <https://lwn.net/Articles/537422/>.
- [26] Tmpfs. <https://www.kernel.org/doc/html/latest/filesystems/tmpfs.html>.
- [27] Top-tier memory management. <https://lwn.net/Articles/857133/>.
- [28] Using DAMON for proactive reclaim. <https://lwn.net/Articles/863753/>.
- [29] zram. <https://www.kernel.org/doc/Documentation/blockdev/zram.txt>.
- [30] N. Agarwal and T. F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. *SIGPLAN*, 2017.
- [31] 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.
- [32] 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.
- [33] 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.
- [34] T. D. Doudali, S. Blagodurov, A. Vishnu, S. Gurumurthi, and A. Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In *HPDC*, 2019.
- [35] J. Du and Y. Li. Elastify cloud-native spark application with PMEM. Persistent Memory Summit, 2019.
- [36] S. R. Dullloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan. Data tiering in heterogeneous memory systems. In *EuroSys*, 2016.
- [37] 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.
- [38] 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.
- [39] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.
- [40] D. Hansen. Migrate pages in lieu of discard. <https://lwn.net/Articles/860215/>.
- [41] B. Holden, D. Anderson, J. Trodden, and M. Daves. *HyperTransport 3.1 Interconnect Technology*. 2008.
- [42] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. 2017.
- [43] 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.
- [44] J. Kim, W. Choe, and J. Ahn. Exploring the design space of page management for Multi-Tiered memory systems. In *USENIX ATC*, 2021.
- [45] 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.
- [46] S.-H. Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, 2016.
- [47] Y. Lee, H. A. Maruf, M. Chowdhury, A. Cidon, and K. G. Shin. Mitigating the performance-efficiency tradeoff in resilient memory disaggregation. *arXiv preprint arXiv:1910.09727*, 2020.
- [48] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang, and O. Mutlu. Utility-based hybrid memory management. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017.
- [49] C. A. Mack. Fifty years of moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 2011.

- [50] H. A. Maruf and M. Chowdhury. Effectively Prefetching Remote Memory with Leap. In *USENIX ATC*, 2020.
- [51] 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.
- [52] 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.
- [53] 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.
- [54] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for nvm+dram hybrid main memory. In *HotOS*, 2009.
- [55] M. Oskin and G. H. Loh. A software-managed approach to die-stacked dram. In *PACT*, 2015.
- [56] 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.
- [57] L. E. Ramos, E. Gorbato, and R. Bianchini. Page placement in hybrid memory systems. In *ICS*, 2011.
- [58] 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.
- [59] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *OSDI*, 2020.
- [60] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, 2018.
- [61] A. Sriraman and A. Dhanotia. *Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale*. 2020.
- [62] A. Sriraman, A. Dhanotia, and T. F. Wenisch. SoftSKU: Optimizing server architectures for microservice diversity @scale. In *ISCA*, 2019.
- [63] Vladimir Davydov. Idle Memory Tracking. <https://lwn.net/Articles/639341/>.
- [64] M. Vuppapapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.
- [65] C. Wang, H. Ma, S. Liu, Y. Li, Z. Ruan, K. Nguyen, M. D. Bond, R. Ne-travali, M. Kim, and G. H. Xu. Smeru: A Memory-Disaggregated managed runtime. In *OSDI*, 2020.
- [66] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen. Exploiting program semantics to place data in hybrid memory. In *PACT*, 2015.
- [67] K. Wu, J. Ren, and D. Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *SC*, 2018.
- [68] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee. Nimble page management for tiered memory systems. In *ASPLOS*, 2019.
- [69] L. Zhang, R. Karimi, I. Ahmad, and Y. Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. 2020.
- [70] 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.