

# GPAC: Memory Tiering with Guest Physical Address Space Consolidation

## Abstract

Memory tiering is the norm to effectively tackle the increasing server memory total cost of ownership (TCO) and the growing data demands of modern data center workloads. However, the **host-based** state-of-the-art memory tiering solutions can be **inefficient** for a virtualized environment when (i) the frequently accessed **data are scattered across the guest physical address space** or (ii) the **accesses to a huge page inside the guest are skewed** due to a small number of subpages being hot. Scattered or skewed accesses make the whole huge page look hot in the host address space. This results in host selecting and placing sparsely accessed huge pages in near memory, wasting costly near memory resources.

We propose GPAC, a **host-agnostic** technique **employed inside the guest** that exploits the two-level address translation in a virtualized environment to consolidate the scattered and skewed accesses to a set of guest physical address ranges. Consolidation transforms sparsely hot huge pages to densely hot huge pages in the host address space context. As a consequence, **host-based tiering solutions can place densely hot huge pages in near memory**, improving near memory utilization.

Our evaluation of GPAC on standalone real-world benchmarks with state-of-the-art host-based tiering shows 50–70% reduction in near memory consumption at similar performance levels, while evaluation at scale improves performance by 10–13% with similar memory TCO.

## 1 Introduction

Memory accounts for 33–50% of the total cost of ownership (TCO) in modern data centers [6, 45]. This cost is expected to escalate further to serve the growing data demands of modern data hungry applications [11, 35, 41]. To effectively tackle growing data demands, it is now a norm in production data centers to employ memory tiering [12, 21, 24, 25, 27, 36, 37, 45].

In memory tiering, hot or frequently accessed pages are periodically identified and placed in faster “near memory” such

as DRAM [46] or HBM [18] while the cold pages are placed in a slower “far memory” such as CXL-attached memory [42] or NVMMs (e.g., Intel’s Optane DC PMM [5, 14]). As near memory resources are capacity limited and costly, a tiering solution [12, 21, 24, 25, 27, 36, 37, 45] dynamically manages placement of data pages across tiers to strike the best balance between performance and memory cost savings

In virtualized cloud environments, **memory tiers are not typically exposed to the guest** [26, 40, 49]. Memory tiering techniques employed in the host identify hot and cold guest pages by tracking memory accesses of individual guest instances in the host context and place them in near and far memory tiers.

We make the following observations for host-based memory tiering in a virtualized environment: First, for a skewed hot huge page [25, 26] where a small portion of a huge page is frequently accessed inside the guest, the entire huge page appear as hot in the host context. As memory tiering techniques are based on page hotness, skewed hot huge pages are placed in near memory leading to under-utilization of costly near memory resources [26, 40].

Second, Cloud Service Providers (CSPs) typically enable huge pages in the host, but the customers buying the guest instances have the freedom to use either huge pages or 4 KB base pages. When base pages are used inside the guest, frequently accessed pages can be scattered in the guest physical address space (GPA). But as GPA directly corresponds to host virtual address (HVA) and HVA is tracked by memory tiering techniques for identifying hot pages, this renders many huge pages hot in the host context (details in §3).

As shown in Figure 1, a skewed hot page in Guest-1 is mapped to a huge page on the host, making it appear hot to the memory tiering technique employed in the host. Similarly, when the guest employs base pages, the scattered hot base pages in Guest-2 maps to huge pages in the host, thereby making them all appear hot in the host context. Since the host typically does not have insight into the guest execution context, it **cannot distinguish between an actual hot huge page and a skewed hot huge page**.

As a result, the host tiering techniques place the entire huge page backing the guest physical address in near memory tier even though a small portion of the huge page is hot or frequently accessed. Hence, a lot of cold subpages that are part of skewed hot huge pages are unnecessarily placed in near memory. This wastes the limited and critical near memory resources and moreover steals the opportunity of placing hot huge pages of other guest instances in near memory.

A naive solution to avoid placing cold subpages in near memory tier is to split huge pages to base pages in the host context and place hot base pages in near memory. Although splitting of huge pages in host is effective for applications running on bare metal [25], it is inefficient in the virtualization environments. Because host cannot identify cold subpages as page accesses are tracked at huge page granularity in the host. In addition, as host in a public cloud environment is unaware of the guest context, it is difficult for host to decide whether to split a huge page or not. A prior solution [26] thus proposed to co-ordinate and split the huge pages both in the guest and the host. But we demonstrate that the skewed hot page problem can be solved by avoiding splitting of huge pages in the host.

We propose GPAC a novel technique that augment memory tiering techniques in virtualized environments. The aim of GPAC is to reduce near memory resources consumed by skewed hot page and improve near memory utilization by transforming skewed hot huge pages to dense hot huge pages by performing guest physical address space consolidation. Consolidation increases densely hot huge pages in the host context which automatically enables memory tiering techniques in the host to mostly place the hot guest pages in near memory. This reduces under-utilization of costly near memory and also reduces per-VM near memory consumption without significantly impacting the performance. GPAC exploits two-level address translation in a virtualized environment to achieve this.

GPAC is host-agnostic, does not require modification to the hardware, and does not require any interactions between the guest and the host. Hence, host can employ memory tiering technique of its choice and continue tiering at huge page granularity.

Importantly, GPAC results in a “win-win” situation for both guest users and CSPs – the guest users experience enhanced application performance without needing to allocate additional memory resources, thus saving cost. While at the host level, GPAC reduces per-guest near memory usage, allowing CSPs to pack pages of many more guest instances in near memory resulting in performance improvement at scale (§5.3).

Our evaluation of GPAC on a single VM with real-world benchmarks with state-of-the-art host-based tiering techniques shows 50–70% reduction in near memory consumption with no loss in performance. Evaluation at scale (with multiple VMs) improves performance 10 – 13% with similar memory TCO.

## 2 Background and Related Work

### 2.1 Two Dimensional Address Translation

The translation of a virtual address (VA) to a physical address (PA) in a native or bare metal setting, requires the walking of the process’s page table in case of a TLB miss. However, in a virtualized environment, a 2-dimensional (2D) walk or nested walk is required to convert the *guest virtual address* (GVA) to the *host physical address* (HPA) as shown in Figure 1. The translation  $GVA \xrightarrow{GPT} GPA$  occurs during the first walk of the *guest page table* (GPT). The  $HVA \xrightarrow{HPT} HPA$  translation occurs during the second level of the walk of the *host page table* (HPT). In virtualized environments such as QEMU/KVM [22], the GPA has a one-to-one linear mapping with the *host virtual address* (HVA). Hardware-assisted virtualization solutions like Extended Page Table (EPT) [2] or Rapid Virtualization Indexing (RVI) [44] were developed to reduce address translation overhead. In the worst case, 24 memory accesses might be required to translate a single GVA to HPA [29].

**Translation with huge pages:** A huge page is a collection of contiguous base pages like 4 KB that improves performance by increasing the TLB reach [15, 26]. The size of supported huge pages is dependent upon *hardware* (CPU) and operating system. For example, Intel x86 CPUs support huge pages of size 2 MB and 1 GB [15]. Moreover, using a huge page brings additional benefits in terms of reducing TLB misses and eliminating the last level of the page table walk in the extended page table (or EPT) in a virtualized environment. With 2 MB huge pages at both guest and host levels, the total number of memory accesses to the page table in the worst case drops to 15 [29].

### 2.2 Identifying Hot and Cold Pages

There are several standard ways to collect telemetry about accessed pages, such as access count, read/write operation, and amount of data used by an application. State-of-the-art telemetry techniques either rely on page table entries [1, 16, 34] or hardware counters [32] to collect the relevant information.

**Page table scanning:** In these methods, the page table entry bits such as *ACCESSED* (*A*) and *DIRTY* (*D*), are manipulated to collect the telemetry. The first two bits, *A* and *D*, are set automatically by the hardware upon a page table walk [31]. A software daemon periodically clears these bits and checks them again after a certain time window. If these bits are set, that indicates that the page was used; otherwise, it was not. Techniques such as Damon [34], idle page tracking (or IPT) [3], Telescope [31] and other [10, 23, 26, 27] are widely used which rely on page table entry bits for their working.

**Hardware counters:** In hardware-based telemetry, the hardware collects statistics about the memory access pattern based

on certain events such as `LOAD` and `STORE` instructions. It collects the corresponding virtual memory addresses and reports them to the OS or the software. Intel PEBS [32] is a hardware-based mechanism to collect the memory access pattern of an application and has been used by different memory tiering solutions [21, 25, 36].

## 2.3 Memory Tiering

Memory tiering is a widely adopted solution to address the growing cost of memory in data centers. As discussed earlier, CSPs employ a heterogeneous memory system with a fast, costly, and small-capacity memory tier such as DRAM or HBM, and a slow, cost-effective, and large-capacity memory tier such as CXL-attached memory [30, 42] and NVMM [14] – henceforth referred to as *near memory* and *far memory*, respectively.

A memory tiering solution dynamically places the data across these different memory tiers based on the access pattern of the application such that most of the LLC misses are served from a *near memory* while ensuring that cold data is placed on *far memory*. There is a plethora of memory tiering solutions proposed by the academia and industry [12, 21, 24, 25, 27, 36, 37, 45]. These tiering solutions differ based on three parameters: ❶ the technique used to collect the telemetry on data page accesses (§2.2), ❷ the choice of memory tiers, and ❸ the policy used to promote or demote a page from a slow tier to a fast tier or vice versa.

Prior work has also explored various memory technologies that can be used as *far memory* tiers such as compressed memory [45], NVMM [12, 21, 25, 36, 37], and CXL-attached memories [21, 25, 27]. Different techniques and heuristics are also employed for hot-cold page classification based on the number of accesses [24, 36], the number of faults [1, 21, 27], or a combination of different events [12].

**Memory tiering in virtualized environment:** In a typical cloud setup, memory tiering solutions are employed at the host level, as the information about the memory tiers is not exposed to guests. Memory tiering mechanisms use HVA to migrate pages across different memory tiers in a virtualized environment.

## 3 Motivation

### 3.1 Access Skewness in a Huge page

Huge pages relieve pressure on the TLB and on different page caching structures [17, 29]. It is also recommended to use huge pages in a virtualized environment (inside guest and host) to improve application performance, as they reduce TLB pressure and Extended Page Table (EPT) walks on TLB misses [17, 29]. However, using huge pages poses challenges in terms of efficient memory management. Specifically, the issue of “skewed hotness,” where a few hot base pages will

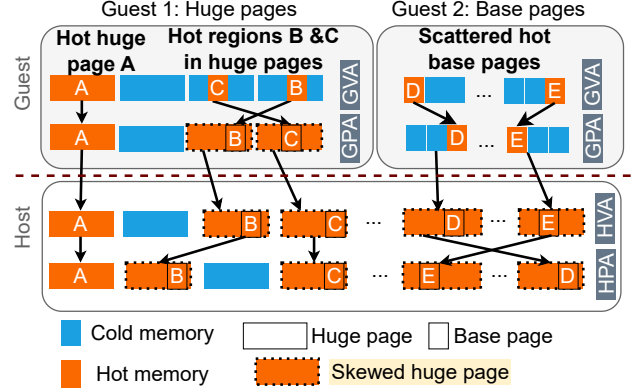


Figure 1: Figuring showing address translation in a virtualized environment and the issue of *Skewed* hot huge pages at the guest and host.

make the entire huge page hot [25]. A hot huge page can be considered *skewed* if the number of hot base pages is less than a *skewness threshold*. In a virtualized environment, a huge page in the host context can be skewed either due to a ❶ skewed hot page or ❷ scattered hot base pages in the guest – as both are mapped to a huge page at the host.

Figure 1 explains skewness in details with an example of two guests, Guest-1 and Guest-2. The host always uses huge pages, whereas Guest-1 uses huge pages and Guest-2 uses base pages. Guest-1 frequently accesses a huge page A and two base pages that are part of huge pages B and C. Such an access pattern generates three hot huge pages at the guest and consequently at the host as well. Because the huge pages in Guest-1 are mapped to huge pages at the host level. Out of the three identified hot huge pages, two (due to hot base pages B and C) are *skewed*. Similarly, Guest-2 frequently accesses two base pages, D and E, which are *scattered* in both guest virtual and physical address space. These two base pages generate two hot huge pages at the host as the base bases in Guest-2 are mapped to huge pages at the host. Both of the identified hot huge pages are *skewed*.

Hence, the host has a total of five hot huge pages where four of them are *skewed* huge pages and one is the *actual* or dense hot huge page (huge page A in Guest-1). As a result memory tiering solutions employed in the host places all the five identified hot huge pages in the *near memory* tier wasting precious *near memory* resources.

To analyze the amount of skewed huge pages generated in real applications we ran experiments with a set of real-world benchmarks and measured the skewness. We observed that only a small portion of a 2 MB huge page is accessed frequently, while the rest of them are not accessed frequently (see Figure 2). For example, in Memcached [7] (with Memtier benchmark [4]), for  $\approx 85\%$  of 2 MB huge pages, less than 100 out of 512 base 4 KB pages are accessed. Hence due to *skewness*, 85% of the total 2 MB huge pages are marked hot

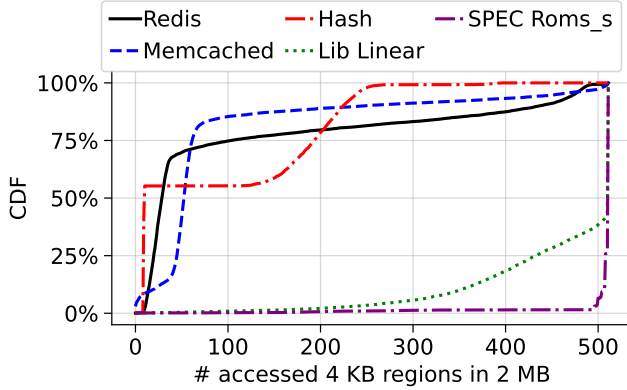


Figure 2: CDF of the number of accessed 4 KB base pages within a 2 MB huge page for a set of real-world workloads.

even though less than 400 KB of data within those individual huge pages are hot. However, there are workloads that do not exhibit skewness such as Liblinear [13] and SPEC 654.Roms [9] where most of the huge pages are densely hot as shown in Figure 2.

In summary, skewed hot huge pages are prominent in many real-world benchmarks. Hence, it is imperative to analyze and handle the issues caused by them in both bare metal and virtualized environments. Prior works [25] have addressed the issue of skewness only in bare metal systems, while the focus of our work is to address access skewness in a virtualized environment.

### 3.2 Inept Memory Tiering due to Skewness

Memory tiering techniques employed in host identify hot data pages of all the guest instances and place them in the fast memory tier, such as DRAM or HBM. As memory tiering solutions cannot differentiate between a dense hot huge page and a skewed hot huge page, they may place skewed hot huge pages in the limited-capacity and costly *near memory* tier. Our empirical evaluation using Redis [8] confirms this where we observed that  $\sim 87\%$  of skewed hot huge pages were placed in *near memory* tier wasting *near memory* resources.

Hence, it is essential to handle skewed hot huge pages in a way that can augment existing host-based memory tiering techniques that are already employed in production environments [1, 12, 27, 45].

### 3.3 Limitations of State-of-the-art

Prior work [15] proposes to demote a skewed hot page to base pages to solve the skewed hotness issue. All subsequent operations, such as memory tiering, are performed at the base page granularity. This solution is widely used both in the host [25] and guest [26] at the cost of increased TLB misses. In a virtualized environment, the cost of such solutions is even

Table 1: The difference in the cost of a TLB miss as reported by Merrifield and Taheri [29] when the host uses 4 KB pages vs. when using 2 MB pages with the guest using 4 KB pages in the both the cases. Naming convention: **Host-{2M/4K}:Guest-{2M/4K}**. Cost is normalized to the H-2M:G-2M setting.

Setting	32 GB	64 GB	256 GB
<b>H-4K:G-4K</b>	5.2 $\times$	4.2 $\times$	4.1 $\times$
<b>H-2M:G-4K</b>	3.2 $\times$	2.3 $\times$	2.3 $\times$

higher as the demotion has to be performed at both the guest and host levels [26]. Just demoting the huge page in the guest while retaining the corresponding huge page in the host does not solve the problem as it results in scattered base pages in the guest as discussed before.

Table 1 shows the overhead of a page table walk in terms of walk cycles normalized to the setting where huge pages are enabled in both host and guest for different working sets [29]. We observe that having a huge page at the host with the guest using base pages results in 50% fewer walk cycles compared to when both levels are using base pages

**Aim:** Solve skewed hot page issues in the guest while retaining (i.e., not demoting) huge pages at the host level to reap the benefits of fewer page table walk cycles.

### 3.4 Host and Guest Interaction

A skewed hot page when placed in *near memory* tier results in inefficient resource utilization in the host. Because it limits the number of VMs with real hot huge pages that can be placed in *near memory*. An intrusive solution where either the host or the guest provides hints or directives to each other on to when and when not to use huge pages results in complex iterations and API dependencies between the host and the guest.

An ideal solution is when both the guest and host are free to choose the size of pages based on their preference. This requires developing solutions that transparently handles skewed hot page without necessitating any communication between the host and the guest. In this paper we aim to develop such a solution that results in a “win-win” situation for both guest and the host.

**Aim:** Propose a host-agnostic solution for skewed hot page in the guest that enables a “win-win” situation for both guest and host.

## 4 GPAC Design and Implementation

In this section, we discuss the GPAC’s design goal and scope (§4.1), key idea (§4.2), and design components (§4.3).



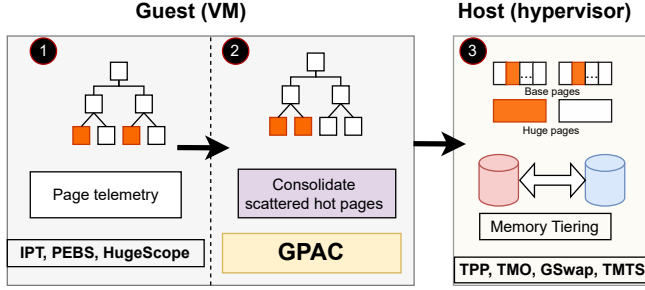


Figure 3: Scope of GPAC. GPAC is compatible with any telemetry solution in the guest and any tiering solution at the host.

#### 4.1 Scope and design goal

Figure 3 shows the high-level approach in solving the challenge of skewed hot pages in a guest on a tiered memory system. Note that the approach assumes that a memory tiering solution is employed at the host level, as the memory tiers are typically not exposed to the guest.

The ❶ first step is to identify skewed hot pages inside the guest. Any state-of-the-art telemetry technique can be used to identify such pages inside the guest. GPAC uses the identified skewed hot pages to take suitable actions. The ❷ second step is to address the skewed hot pages inside the guest. Prior solutions usually demote them at the guest and host level. GPAC addresses this by consolidating pages in guest physical address space. The ❸ third step is to employ memory tiering techniques in the host.

**Scope of GPAC:** GPAC operates at the second step and mitigates skewed hot page at the host. The input to GPAC is a set of telemetry data at base page granularity. GPAC is responsible for identifying skewed hot page based on the telemetry data.

Efficient tracking of hotness (telemetry) in a guest is *not* in the scope of this work, and any existing state-of-the-art technique can be used with GPAC [16, 26]. Furthermore, the host is free to use any memory tiering technique. We assume that hosts always use huge pages to benefit from the higher TLB reach [23, 38, 43].

**Design goals:** We design GPAC to achieve the following goals:

1. **Host-agnostic:** Develop a solution that does not need any modification in the host or hypervisor (§5.3 and §5.4).
2. **Tiering technique agnostic:** Any existing memory tiering solution can be used at the host without requiring any modifications (§5.3).
3. **Memory tier agnostic:** Should work seamlessly on any hardware memory tiers. Any memory technologies, such as DRAM, Intel Optane, CXL-attached memory, High Bandwidth Memory (HBM), can be used as memory tiers (§5.4).

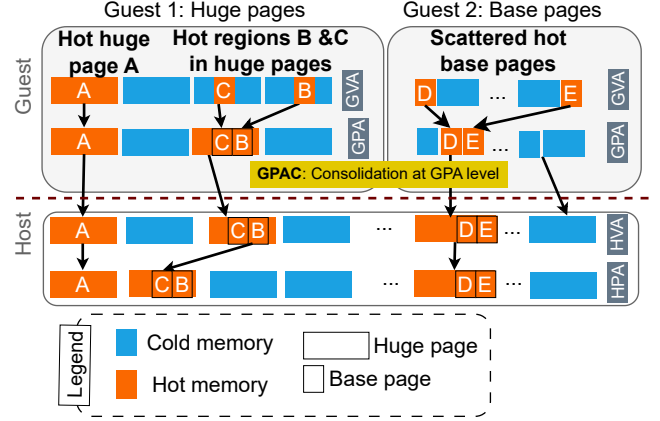


Figure 4: Illustration of hot base page consolidation inside the guest to reduce the number of *skewed* hot huge pages at the host.

4. **Telemetry agnostic:** Any telemetry tools (mentioned in §2.2) to identify hot and cold pages can be used without any modification.
5. **Leverage huge pages:** Get optimal benefits of TLB by avoiding splitting of huge pages at the host.

#### 4.2 Key idea

GPAC exploits the additional level of address translation in a virtualized environment inside the guest ( $GVA \xrightarrow{GPT} GPA$ ) to “consolidate” scattered hot base pages into a huge page size region in the guest backed by a huge page in the host

**Consolidation:** Consolidation refers to the process of gathering all scattered hot base pages to a single huge page-sized region inside the guest. Consolidation operates inside the guest at GPA level and comprises of data copy of scattered hot base pages and updating the  $GVA \xrightarrow{GPT} GPA$  mapping accordingly. The process of consolidation reduces the number of skewed hot pages at the host without any modification at the host, as only the mappings  $GVA \rightarrow GPA$  are updated.

Using the same example from earlier (§3.1), we explain GPAC’s key idea (see Figure 4). The process of consolidation moves the content of hot base pages *B* and *C* together in a huge page region, which is backed by a huge page at host and modifies  $GVA \rightarrow GPA$  mappings inside Guest-1. Similarly, pages *D* and *E* inside Guest-2 are consolidated to a single huge page region. After consolidation, all skewed hot pages are eliminated, resulting in three densely hot huge pages in the host context (see Figure 4). Hence consolidation is reducing hot pages from earlier five (as seen in Figure 1) to three. Hence, memory tiering solutions at the host will place these three densely hot huge pages in *near memory*.

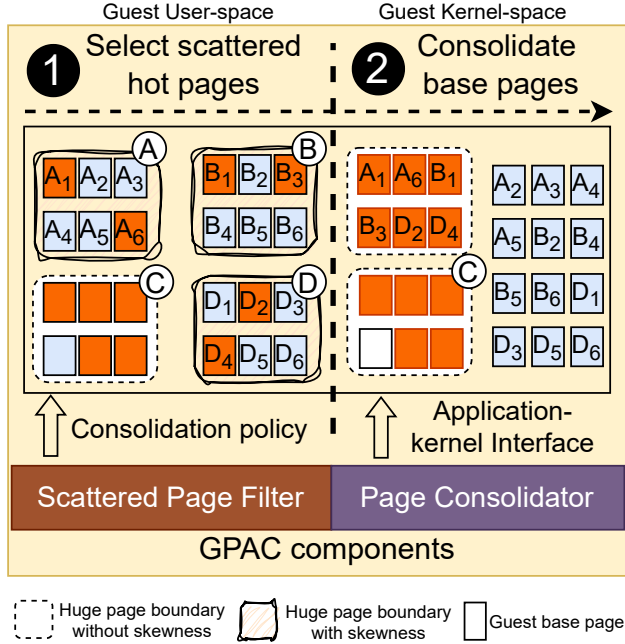


Figure 5: Design components of GPAC for memory consolidation inside guests.

### 4.3 Design components

GPAC operates through multiple layers (guest user space and guest kernel space) as shown in Figure 5 and consists of two major components inside the guest: *Scattered Page Filter* and *Page Consolidator* as shown in Figure 5.

#### 4.3.1 Scattered Page Filter

As mentioned in Section 4.1, efficient telemetry is *not* in the scope of GPAC. Hence any existing state-of-the-art telemetry technique can be used with GPAC to find *scattered hot pages*.

The input to GPAC is a list of hot base pages ( $L_{scatter}$ ). These pages can be either scattered in the guest physical address space or can be within a huge page region. *Scattered Page Filter* operates in the guest's user space to select hot base pages from  $L_{scatter}$  to consolidate as per the Consolidation policy as shown in Figure 5.

**Consolidation policy in GPAC:** We can reduce the number of skewed hot pages at host by consolidating hot base pages inside the guest. However, consolidation is not free as it incurs performance overhead due to data copy and page table updates inside the guest.

We provide a user-tunable parameter – *Consolidation Limit* (or CL) to balance consolidation opportunity and application performance. *Scattered Page Filter* uses CL as a threshold limit to filter out pages that need to be consolidated. For example, a 2 MB huge page contains 512 base pages of size 4 KB. If the user provides CL as 20, then *Scattered Page Filter*

#### Algorithm 1 Page Consolidation: `consolidate_pages()`

**Require:** 1.  $PAGE\_SIZE$  = Size of Base page

2.  $HPAGE\_SIZE$  = Size of huge page

3.  $MAX\_CONSOL = HPAGE\_SIZE/PAGE\_SIZE$

4.  $L$ : List of scattered hot pages,  $1 \leq |L| \leq MAX\_CONSOL$

5.  $pid$ : Process id of the target process

**Ensure:** Consolidated to a single huge page region

1: **Initialization:**

2:  $huge\_region\_ptr \leftarrow page\_alloc(HPAGE\_SIZE)$

3: **if**  $huge\_region\_ptr$  is NULL **then**

4:     return -ENOMEM

5: **end if**

6:  $mm$  = Memory descriptor ( $mm\_struct$ ) of  $pid$

7: **For each** page  $old\_page$  **with index**  $i$ :

1.  $new\_page \leftarrow huge\_region\_ptr + i * PAGE\_SIZE$

2.  $lock\_page(old\_page)$      ▷ Prevent concurrent modification

3.  $lock\_page(new\_page)$      ▷ Prevent concurrent modification

4.  $addr$  = virtual address of the  $old\_page$

5. **migrate\_page\_move\_mapping()**

(a)  $memcpy(new\_page, old\_page)$      ▷ Actual page content copy

(b)  $pte\_t\ new\_pte = mk\_pte(new\_page, \dots)$ ; ▷ Prepare page table pointer

(c)  $ptep = get\_ptep(addr)$      ▷ get ptep for address  $addr$

(d)  $set\_pte\_at(mm, addr, ptep, new\_pte)$      ▷ Update page table entry to map  $addr$  to  $new\_page$

(e)  $flush\_tlb\_mm\_range(mm, addr, addr + PAGE\_SIZE)$      ▷ TLB flush

6.  $unlock\_page(new\_page)$

7.  $unlock\_page(old\_page)$

8.  $free(old\_page)$

will select only those huge page region that have less than 20 hot base pages of size 4 KB. A lower value for CL will consolidate fewer scattered pages, whereas a higher value of CL will perform an aggressive consolidation to reduce number of hot huge pages at the host but at the cost of data copy and TLB update overheads.

**Implementation:** GPAC uses the Idle Page Tracking tool [3] (or IPT), as the telemetry technique inside the guest to track access to data pages in a given time window. Note that IPT operates in the user space and provides virtual addresses at page size granularity. *Scattered Page Filter* uses the set of pages reported by IPT and filters out hot pages for consolidation based on the *consolidation limit* (CL) (§4.3.1). The set of filtered pages that needs to be consolidated is passed on to the page consolidator implemented in the guest kernel.

#### 4.3.2 Page Consolidator

*Page Consolidator* operates in the kernel space of the guest to consolidate base pages by moving base pages to huge page-sized region(s) in the guest. For example, a maximum of 512 4 KB base pages can be consolidated in a single huge page of size 2 MB. Page consolidator exposes an application-kernel interface to invoke the consolidation of the base pages selected by page filter described before (Figure 5). Consol-

idator reserves a contiguous memory of size equal to a huge page and moves the contents of selected base pages from the filtered hot base page list to the newly allocated huge page-sized region. The consolidator also updates the GVA to GPA mappings. The freshly allocated region ensures that the filtered base pages are contiguous at the GPA level and, naturally, at the HVA level and hence can be backed by a huge page in the host.

**Implementation:** *Page Consolidator* exposes application-kernel interface to get necessary information from the guest user space, such as process `pid` and a set of hot pages to be consolidated using a custom system call “`consolidate_pages()`”. `consolidate_pages()` system call takes a list of scattered pages that need to be consolidated in the guest address space and moves them to a huge page-sized region using steps mentioned in Algorithm 1. `consolidate_pages()` consolidates a maximum number of 512 base pages per invocation. Multiple invocations are required if more base pages should be consolidated. For example, a huge page of size 2 MB can accommodate a maximum of 512 4 KB base pages and `consolidate_pages()` needs to be invoked twice to consolidate 1024 base pages.

## 5 Evaluation

### 5.1 Experimental setup

**Host:** Our experiment setup is an Intel Xeon Gold 6252N CPUs running at 2.30 GHz having 2 sockets. Each socket has 48 cores with hyper-threading and contains a total of 375 GB DRAM and 1.6 TB of NVMM (Intel Optane). From the OS perspective, the CPUs and memory show up as 4 NUMA nodes, with 2 nodes containing  $\approx 188$  GB of DRAM and 48 CPUs each and the other two nodes with  $\approx 825$  GB of NVMM memory each. The host uses Linux kernel 5.18 and always uses huge pages.

For experiments with HBM and CXL, we use a two-socket system with two Intel(R) Xeon(R) Max 9480 CPUs with each socket containing 64 GB of HBM memory, 1 TB of DRAM memory, and 256 GB of CXL-attached memory.

**Guest:** We use the KVM hypervisor [22] to run guest(s) with the Linux kernel 5.18. Each guest runs a single instance of the application, and it is configured with 20 GB of main memory and 12 vCPUs. We restrict guests to using only 4 KB base pages as telemetry is not the main goal of GPAC, and skewed hot pages are identified only after they are demoted to base 4 KB pages [26]. Restricting to base pages allow us to use an off the shelf telemetry and focus on the main contribution of the work.

**Workloads:** We evaluated the effectiveness of GPAC using a set of a microbenchmark and several real-world workload as shown in Table 2 along with their guest *resident set size* or RSS. We configured Masim [33] workload to access only one

Table 2: Benchmarks and their description with memory footprint (guest RSS).

Workload	Description	Guest RSS
Masim [33]	Memory access simulator	9.8 GB
Redis [8]	In-memory key-value store	12.5 GB
Memcached [7]	In-memory key-value store	11 GB
Hash [28]	Hash datastructure	8.8 GB
Ocean_ncp [48]	Ocean simulation	5.5 GB

4 KB page out of 512 pages in a huge page boundary. For Redis [8] and Memcached [7], we populate data with a key size of 1 KB and use the Memtier [4] as a workload generator [4] to generate requests as per a Gaussian distribution. We host both the server and client (Memtier) within the same guest to prevent any potential network bottlenecks. For the Hash workload [28], we used Hash\_bkt\_rcu [28] which is a hash table protected by a per-bucket lock for updates and RCU for lookups. We also use Ocean\_ncp which is an Ocean simulation with W-cycle multigrid solver, SPLASH2x application from PARSEC 3.0 [48].

**Memory tiering at host:** We configure the host with state-of-the-art memory tiering mechanisms such as AutoNUMA [1], TPP [27], and Memtierd [20] (one at a time). AutoNUMA and TPP are kernel-level memory tiering mechanisms. However, Memtierd operates in the user space to perform memory tiering.

**Telemetry inside guest:** As discussed earlier (§2.2), several tools exist to determine the hot and cold memory pages, such as Intel PEBS [32], DAMON [34], and Idle Page Tracking [3]. Our work is independent of the telemetry technique used to detect skewness hotness inside a guest. Our goal is not to optimize detection of skewed hotness inside a guest, and any existing techniques [16, 26, 32, 40] can be used with GPAC. For our prototype, we use the widely used Idle Page Tracking [16] inside guest to find hot and cold pages.

### 5.2 Evaluation: single guest instance

We evaluate the effectiveness of GPAC with a single guest with no near memory pressure. We use Memtierd for memory tiering as it can perform tiering even without any memory pressure. We start the VM using only far memory tiers as the *preferred* memory tier, and the host performs memory tiering and migrates identified hot pages to the near memory.

#### 5.2.1 Hotness consolidation at host

As discussed earlier, the opportunity to reduce the number of hot huge pages at the host depends upon selected scattered hot base pages in the guest as per the tunable parameter CL, and also on the number of available contiguous memory regions of size equivalent to a huge page. Table 3 shows the CL, selected hot base pages, and the time taken to perform consol-

Table 3: Consolidation time of selected hot 4 KB pages using CL for different workloads.

Workload	Consol. limit	Selected 4 KB pages	Consol. Time (ms)
Masim	10	4,142	36
Redis	50	93,896	840
Memcached	100	174,068	1220
Hash	250	307,484	3,363
Ocean_ncp	290	950,758	7,329

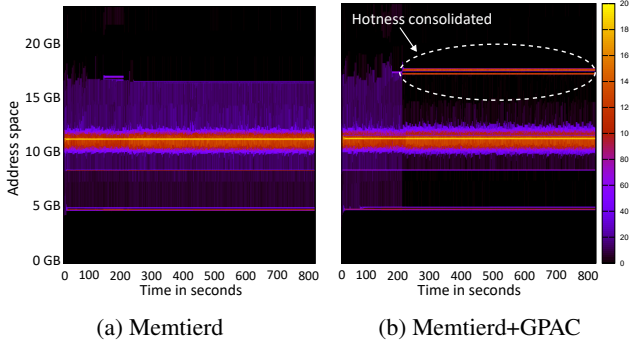


Figure 6: Figure showing detected hot region with Redis at host using DAMON [34]. GPAC consolidates scattered hot base pages into a few huge page regions.

idation. We successfully consolidated all selected hot 4 KB pages for all the workloads. We used different CL for different workloads based on workload’s memory access pattern.

**Discussion:** We do a deep dive with Redis to understand the consolidation of hot huge pages at the host level. Figure 6a shows the identified hot regions at the host level as reported by the DAMON tool [34]. We see a hot region around the 11 GB offset in the address space, which decreases in intensity as we move away from this hot region. Figure 6b shows the heatmap after hot pages consolidation using GPAC. We observe that scattered hot regions are getting consolidated into smaller regions with an increased degree of hotness around the 16 GB offset in the address space.

### 5.2.2 Reduction in near memory usage

We analyzed the efficacy of GPAC in reducing *near memory* usage and its impact on the application running inside the guest. We start Redis with all the allocations from the far memory tier. The memory tiering solution, Memtierd, starts after  $\approx 220$  seconds and migrates  $\approx 85\%$  of data to the near memory tier (see Figure 7a). However, with GPAC which consolidates the skewed hot pages inside the guest, the total amount of data migrated to the near memory tier drops to 33% with no loss in performance (see Figure 7b).

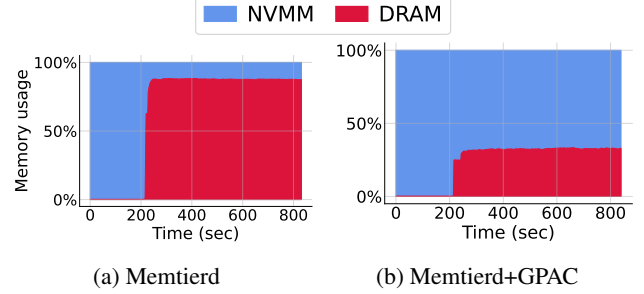


Figure 7: Memory distribution (% of guest RSS) between DRAM and NVMM using Memtierd and Memtierd+GPAC for Redis workload.

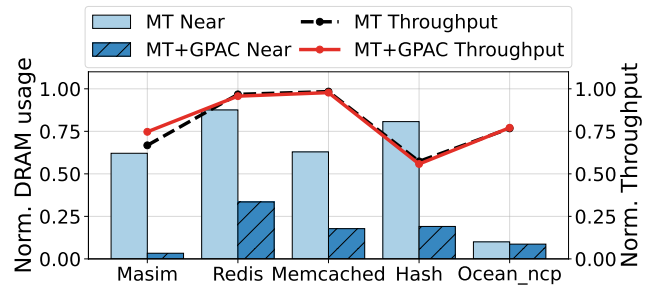


Figure 8: Reduction in DRAM usage with GPAC and tiering with Memtierd (MT) compared with just tiering by Memtierd. GPAC reduces near memory usage with no loss in performance. Normalized to all-DRAM setting without memory tiering.

### 5.2.3 Impact on performance

Figure 8 shows the reduction in near memory usage when using GPAC in the guest and Memtierd-based tiering in the host compared to using only memory tiering at the host. We see an average reduction of 72% in near- memory tier usage for all workloads with a negligible loss in performance of  $\approx 0.86\%$  (excluding the microbenchmark Masim). As GPAC is host agnostic, the tiering solution in the host migrates data completely based on the identified hot/cold pages. As the amount of data has dropped significantly, it indicates the amount of detected hot data has decreased. The reduction in the amount of hot data is due to the consolidation done by GPAC of hot 4 KB base pages into fewer 2 MB densely hot huge pages. Reducing the amount of hot data also helps in reducing the significant cost associated with migrating data pages from one NUMA node to another, negatively impacting the application’s performance [19, 47].

**Summary:** GPAC reduces the number of hot huge pages at the host and required page migration for memory tiering due to the consolidation of hot 4 KB pages inside the guest. As a result, it GPAC reduces the *near memory* usage without significant impact on the application’s performance.



### 5.3 Benefits of GPAC at scale

In the previous section, we demonstrated that GPAC significantly reduces DRAM usage without impacting applications' performance. In this section, we measure the efficacy of GPAC in a multi-tenant scenario when multiple VMs are hosted on a single host machine. In such a setting, DRAM memory used by one VM will impact the performance of other VMs as their LLC misses will be served from the NVMM. However, as GPAC reduces the DRAM usage, it results in a performance improvement across multiple VMs. We configure multiple guests (each hosting one Redis workload with Memtier) on the same host to simulate a DRAM-constrained environment.

**Host configuration:** We configure the host to use Linux kernel version 6.9. We did not bind guests to any memory node and used the default memory placement policy for workloads. We show the performance of GPAC with different memory tiering techniques at the host, such as AutoNUMA with memory tiering enabled<sup>1</sup> [1, 39], TPP [27], and Memtierd [20] (enabled one at a time). Guest configuration remains same as mention in the previous section.

#### 5.3.1 Memory tiering at host using Memtierd

We use Memtierd tiering technique to reduce DRAM memory by relocating cold pages to a far memory tier. Here, we start guests on DRAM and then perform memory tiering to observe the benefits of optimal DRAM usage across guests in a DRAM-constrained environment. We run six guests (each running Redis with 12.5 GB RSS as shown in Table 2) with a fixed size of 20 GB of DRAM and 80 GB of NVMM available for all guests to use.

**DRAM improvement:** As can be observed in Figure 10a, with Memtierd, DRAM was unevenly distributed amongst the host on a first-come-first-serve basis. Due to skewed hot page, majority of data pages in the VMs are identified as hot, and hence, they keep on occupying space in the precious DRAM. VM1 uses an average of 46% of DRAM, whereas the rest of the VMs use 3% - 11% of DRAM, resulting in performance degradation for those VM. With GPAC enabled in all the guests, the number skewed hot page came down at the host, and more cold huge pages are demoted to the NVMM (far memory tier), resulting in creating DRAM space for actual hot huge pages from VMs. This results in a much more even distribution of DRAM usage, with the first five VMs using an average of 18% of DRAM. VM6 uses just 6% of DRAM; however, those are actual hot huge pages instead of skewed hot page.

**Performance improvement:** Due to a better distribution of DRAM usage, we observe that Memtierd+GPAC improves the performance by  $\approx 13\%$  on average, over just Memtierd, as shown in Figure 9a. The performance of all guests showed

an improvement of 12% - 19% with Memtierd+GPAC, compared to Memtierd, with the exception of VM1 as shown in Figure 9a. For VM1, we see a performance improvement of  $\approx 1\%$  even though its DRAM usage is down from 46% with Memtierd to just 18% with Memtierd+GPAC. Because, now only the actual hot huge pages from VM1 are occupying the DRAM. As a result, GPAC provided better overall system performance due to increased hotness of huge pages at the host level and better utilization and distribution of DRAM.

#### 5.3.2 Memory tiering at host using AutoNUMA and TPP

AutoNUMA [1] and TPP [27] are the two state-of-the-art memory tiering solutions available in the Linux kernel. Both the techniques perform memory tiering only under memory pressure by demoting cold pages to far memory tiers and promoting hot pages to the near memory tier. We configured the host with 64 GB DRAM (32 GB each node) and 512 GB NVMM (264 GB each node) and hosted eight guests (each with 12.5 GB RSS as shown in Table 2) to create pressure on DRAM.

**Memory savings:** Figure 11a and Figure 11b show the data promoted from NVMM to DRAM and data demoted from DRAM to NVMM, respectively, with just TPP and TPP+GPAC. We observe the TPP+GPAC reduces the total data promoted and demoted by an average of  $\approx 64\%$  and  $\approx 87\%$ , respectively, due to the consolidation of hot pages inside the guest to a few sets of hot huge pages at the host level. A similar trend is observed for AutoNUMA vs AutoNUMA+GPAC (not shown due to space constraints).

**Performance impact:** In terms of performance, TPP+GPAC improves the performance by an average of  $\approx 11\%$  for all eight guests over TPP as shown in Figure 9b. The performance of VM7 and VM8 improves by 20% and 18%, respectively.

For AutoNUMA, we observe an average performance improvement of 1.6% with AutoNUMA+GPAC over just AutoNUMA as shown in Figure 9c. Out of 8 VMs, 5 VMs show an average performance improvement of 6.6%, whereas three VMs, VM4, VM6, and VM8, incur a performance overhead of 6%. We analyze the performance gains in detail in the next section 5.3.3.

#### 5.3.3 Deep dive

In this section, we perform a deep dive to understand the cause of performance improvement with GPAC in different tiering solutions. We use hardware performance counters [32] to capture a few key statistics, such as loads served from NVMM (`mem_load_retired.local_pmm`), dTLB load misses (`dTLB-load-misses`), and execution stalls due to memory subsystems at the host level (`cycle_activity.stalls_mem_any`). Figure 12 shows the per-VM collected hardware counter stats for Memtierd and TPP, both with GPAC enabled in the guest (AutoNUMA not

<sup>1</sup>`echo true > /sys/kernel/mm/numa/demotion_enabled`  
`echo 2 > /proc/sys/kernel/numa_balancing`

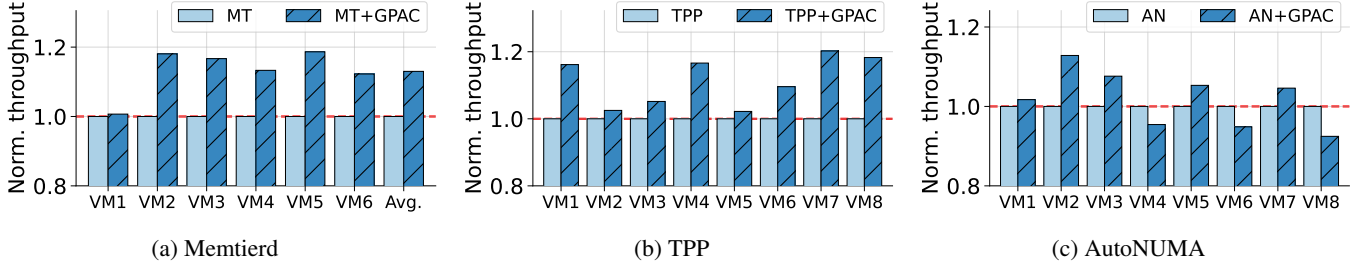


Figure 9: Performance improvement in GPAC while using AutoNUMA, TPP, and Memtierd for memory tiering at host.

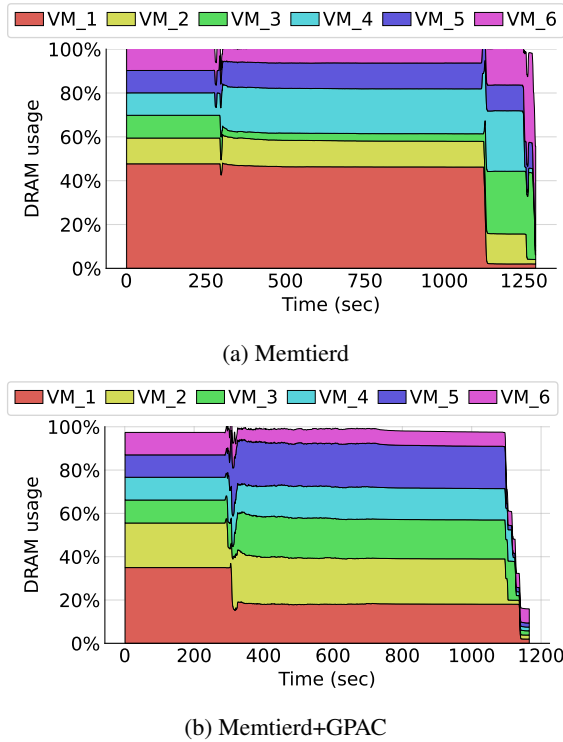


Figure 10: DRAM memory distribution across guests using Memtierd and Memtierd+GPAC.

shown due to space constraints). Note that the hardware performance events are captured at the host level using Intel PEBS [32].

**Using Memtierd at host:** We observed significant performance gain while using Memtierd+GPAC, compared to Memtierd (§5.3.1). PMU events for TPP and TPP+GPAC for each guest are shown in Figure 12a, Figure 12b, and Figure 12c. We observed that Memtierd+GPAC reduces NVMM access and stalls by 90% and 25%, respectively, compared to Memtierd. However, dTLB load misses increase by 3% mainly due to TLB shutdowns associated with consolidation and migration as we start from DRAM and migrate all the cold data to NVMM.

We observed that NVMM access for VM1 remains un-

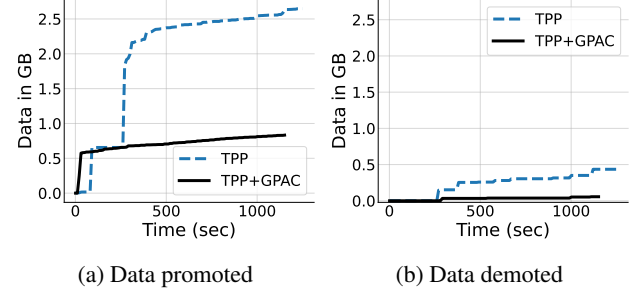


Figure 11: Page migration and demotion using TPP and TPP+GPAC. Memory tiering starts after initialization at around 300 seconds.

changed as it used a significant portion of DRAM with Memtierd. With GPAC +Memtierd only consolidated hot huge pages remain in DRAM, and cold data is moved to NVMM (Figure 10a and Figure 10b). However, NVMM access for remaining guests (VM2, VM3, VM4, VM5, and VM6) was significantly lower in case of Memtierd+GPAC compared to Memtierd due to better utilization of DRAM by only keeping actual hot data in DRAM (Figure 12a).

**Using TPP at host:** PMU events for TPP and TPP+GPAC for each guest are shown in Figure 12d, Figure 12e, and Figure 12f. We observed significant gain in performance while using TPP+GPAC, compared to TPP, as discussed before. We observed that TPP+GPAC reduces NVMM access, and stalls, and dTLB load misses by 82%, 35%, and 74%, respectively, compared to TPP.

Since in TPP, we start from NVMM and migrate to DRAM based on the detected hotness, skewed hot page hog down the precious DRAM resources forcing the accesses to go to the slow NVMM. With TPP+GPAC, the number of NVMM access for 3 VMs comes down by 90%+, with VM3 showing a reduction of 99.41%. This translates to a reduction in stalls by up to 42%. Since, TPP+GPAC migrates far fewer data from NVMM to DRAM, the number of dTLB load misses drops by up to 88%.

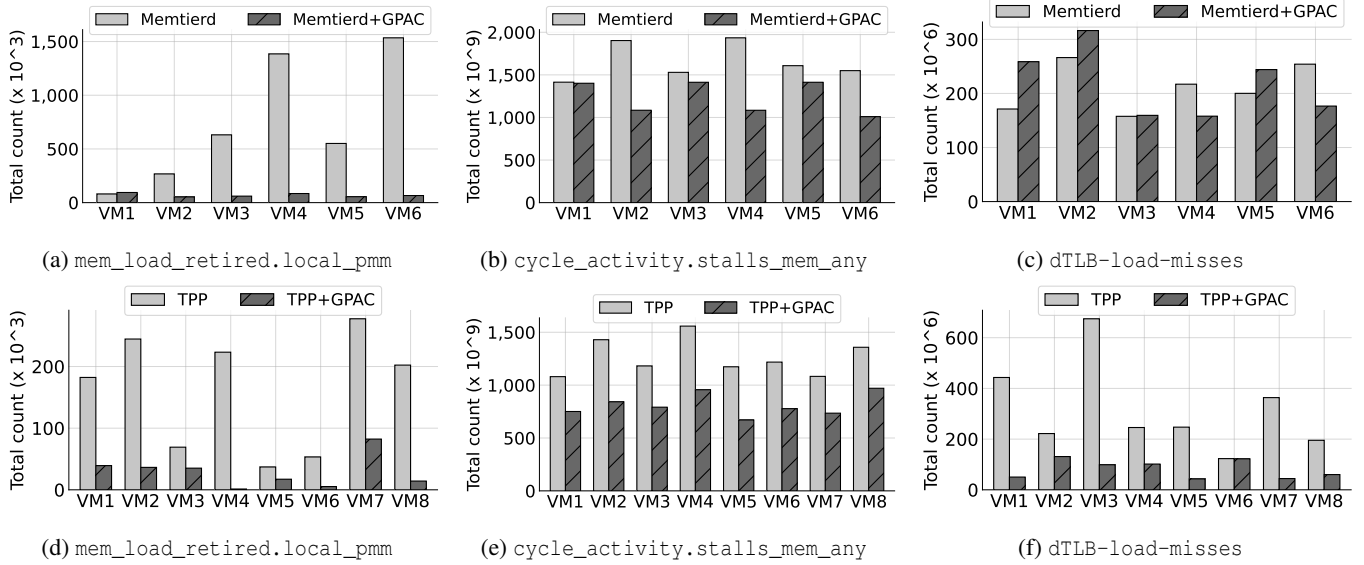


Figure 12: Figures showing the changes in hardware counters for the total number of loads to NVMM, memory stall cycles, and dTLB load misses for Redis workloads with Memtier and TPP, with and without GPAC.

#### 5.4 GPAC with emerging memory technologies

GPAC is a generic approach and is agnostic to employed memory tiers at the host level. We evaluate GPAC with two emerging memory technologies: *High Bandwidth Memory*, or HBM, and *Compute eXpress Link*, or CXL-attached memories. With HBM, DRAM acts as the far memory tier, and HBM as the near memory tier, as HBM offers a lower access latency and higher bandwidth than DRAM [18]. With CXL-attached memories, DRAM acts as the near memory tier and CXL-attached memory as the far memory tier, again as per offered performance. We use a real CXL card [30], which offers an access latency higher than DRAM but lower than that of NVMM memory.

Note that even with the change in the hardware memory tiers, the logic to detect hot/cold pages, the issue of skewed hot page, and the working of GPAC remains the same.

**DRAM & CXL-attached memory:** We configure the host with 30 GB DRAM and 70 GB CXL-attached memory for a fixed 100 GB of total memory. We start six guests, each with a Redis workload with an RSS of 12.5 GB and use Memtier for memory tiering at the host. We observed that Memtier+GPAC resulted in  $\approx 6.3\%$  average performance improvement over Memtier, as shown in Figure 13. We also observed 7.8% and 4.9% reductions in p50 and p99 latencies, respectively. The reduction in latency is due to an increase in the number of memory accesses being served from the DRAM instead of high-latency CXL memory.

**HBM & DRAM:** To analyze the effectiveness of GPAC using HBM, we configured the host with 20 GB of HBM as the near memory tier and 80 GB of DRAM as the far memory tier. We

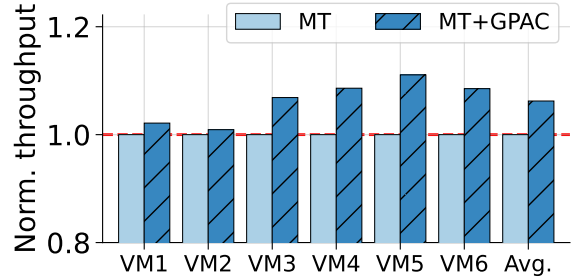


Figure 13: Performance impact with DRAM as the near memory tier and CXL-attached memory as the far memory tier.

observed that Memtier+GPAC resulted in 5.3% average performance improvement over Memtier as shown in Figure 14. We also observed a 7.8% reduction in p99 latency due to low latency offered by the HBM.

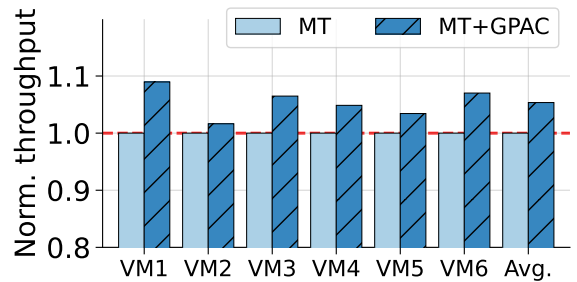


Figure 14: Performance impact with HBM as the near memory tier and DRAM as the far memory tier

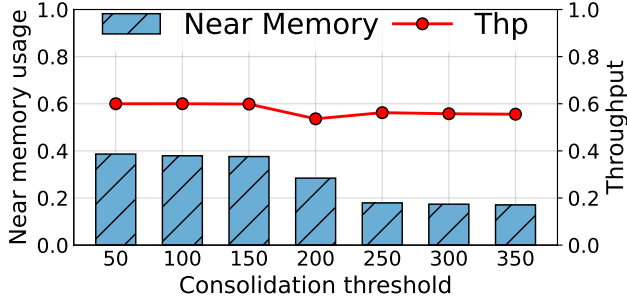


Figure 15: Significance of CL on *near memory* usage and application performance for Hash workload while using Memtierd at host. Normalized to all-DRAM setting without memory tiering.

## 5.5 Sensitivity Analysis

### 5.5.1 Consolidation limit sensitivity

It is required to keep a balance between the consolidation of scattered hot base pages to reduce the number of hot huge pages and application performance. Consolidation limit or CL plays a vital role in keeping balance between mentioned trade-offs – a high value indicates an aggressive consolidation, and thus, a large reduction in the number of hot huge pages in the host. Whereas a low value for CL consolidates a few scattered pages, resulting in a small reduction in the number of huge pages.

Figure 15 shows the performance impact and DRAM memory savings with the hash workload for different values of the consolidation limit. With the increase in the CL (beyond 150), the DRAM usage comes down, with a slight impact on the performance. However, the DRAM savings and performance overhead saturate after CL crossing 250. This can be explained from Figure 16b, where we see a large number of huge pages have only 150 hot base pages (hence, the increase in memory savings with CL set to 150). Furthermore, there are only a few pages with hot base pages more than 250, hence the memory savings saturate with CL as 250.

### 5.5.2 Effectiveness under varying memory pressure

We also performed experiments to analyze impact of GPAC under different *near memory* pressure situation. The total amount of memory remains fixed at 100 GB, but we varied the amount of DRAM to NVMM ratio. Figure 17 shows the average throughput of all six guests for Memtierd and Memtierd+GPAC.

We observed that Memtierd+GPAC performed significantly better than Memtierd for ratios 10:90, 20:80, and 30:70. However, the benefit of Memtierd+GPAC gradually reduces with the increase in the amount of available DRAM memory. Because when more DRAM is available opportunity for tiering optimization reduces. It can also be noted that

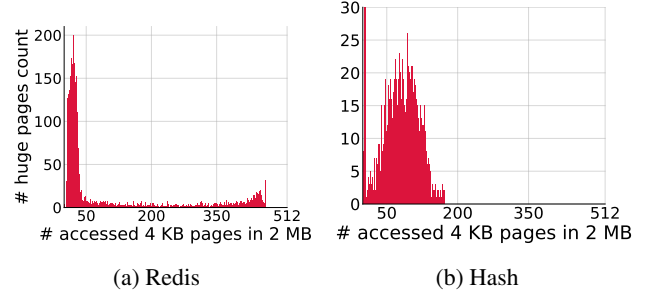


Figure 16: Scattered hot 4 KB pages across several huge pages for different workloads. # count (Y-axis) represents number of huge pages having “k” unique 4 KB page accessed, where “k” is the value on the X-axis (# accessed 4 KB regions in 2 MB).

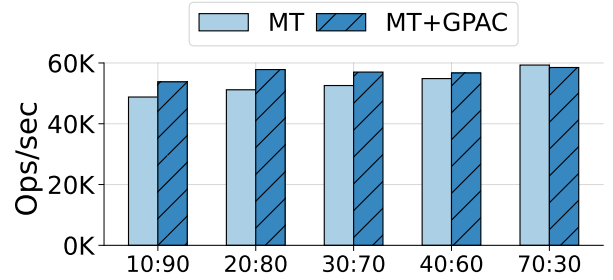


Figure 17: Average throughput (Ops/sec) of six guests under varying *near memory* to *far memory* ratios using Memtierd (MT).

Memtierd+GPAC performance decreases for 70:30. To summarize, Memtierd+GPAC performed better than Memtierd in the *near memory* pressure situation. However, opportunity for improvement decreases when more amount of *near memory* is available as the workload can fit most of the pages including skewed hot pages in DRAM.

## 6 Conclusion

Efficient memory tiering reduces Total Cost of Operation by reducing costly *near memory* usage. In this work, we demonstrated that consolidation of hot pages scattered across several huge pages inside a guest significantly reduces the number of hot huge pages (by increasing hotness) at the host, further reducing the *near memory* usage. We proposed GPAC, a guest physical address space consolidation mechanism, for efficient *near memory* usage without any modification in the existing memory tiering mechanism at host. GPAC improved application performance by upto 10-13% at scale while using existing state-of-the-art memory tiering solutions like TPP, and Memtierd at host.



## References

- [1] AutoNuma: NUMA BALANCING MEMORY TIERING. <https://docs.kernel.org/admin-guide/sysctl/kernel.html#numa-balancing>.
- [2] Performance Evaluation of Intel EPT Hardware Assist. [https://www.vmware.com/pdf/Perf\\_ESX\\_Intel-EPT-eval.pdf](https://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf).
- [3] Idle Page Tracking. [https://docs.kernel.org/admin-guide/mm/idle\\_page\\_tracking.html](https://docs.kernel.org/admin-guide/mm/idle_page_tracking.html).
- [4] Memtier Benchmark. [https://github.com/RedisLabs/memtier\\_benchmark](https://github.com/RedisLabs/memtier_benchmark).
- [5] Intel® optane™ dc persistent memory product brief. <https://www.intel.in/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf>. (Accessed on 08/01/2023).
- [6] Decadal plan for semiconductors. <https://www.src.org/about/decadal-plan/decadal-plan-full-report.pdf>.
- [7] memcached - a distributed memory object caching system. <https://memcached.org/>, 2019. (Accessed on 11/18/2019).
- [8] redis. <https://redis.io/>, 2019. (Accessed on 11/18/2019).
- [9] SPEC CPU 2017. Standard performance evaluation corporation. <https://www.spec.org/cpu2017/>, 2024.
- [10] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 631–644, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344654. doi: 10.1145/3037697.3037706. URL <https://doi.org/10.1145/3037697.3037706>.
- [11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL [https://proceedings.neurips.cc/paper\\_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf](https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf).
- [12] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582031. URL <https://doi.org/10.1145/3582016.3582031>.
- [13] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *the Journal of machine Learning research*, 9:1871–1874, 2008.
- [14] Shivank Garg, Aravinda Prasad, Debadatta Mishra, and Sreenivas Subramoney. Motivating next-generation os physical memory management for terabyte-scale nvmm, 2023.
- [15] Mel Gorman. Huge pages part 1 (introduction) [lwn.net], 2010. URL <https://lwn.net/Articles/374424/>. [Online; accessed 2025-03-16].
- [16] Christian Hansen. Linux idle page tracking, 2018. URL [https://www.kernel.org/doc/Documentation/vm/idle\\_page\\_tracking.txt](https://www.kernel.org/doc/Documentation/vm/idle_page_tracking.txt).
- [17] Weiwei Jia, Jiyuan Zhang, Jianchen Shan, and Xiaoning Ding. Making dynamic page coalescing effective on virtualized clouds. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 298–313, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394871. doi: 10.1145/3552326.3567487. URL <https://doi.org/10.1145/3552326.3567487>.
- [18] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE, 2017.
- [19] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of*

the 44th Annual International Symposium on Computer Architecture, pages 521–534, 2017.

- [20] Antti Kervinen. Memtierd. <https://github.com/intel/memtierd>, December 2024. (Accessed on 12/09/2024).
- [21] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for Multi-Tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 715–728. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL <https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon>.
- [22] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.
- [23] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon>.
- [24] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019. URL <http://doi.acm.org/10.1145/3297858.3304053>.
- [25] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 17–34, 2023.
- [26] Chuandong Li, Sai Sha, Yangqing Zeng, Xiran Yang, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Diyu Zhou. Taming hot bloat under virtualization with HUGESCOPE. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 999–1012, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-41-0. URL <https://www.usenix.org/conference/atc24/presentation/li-chuandong>.
- [27] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582063. URL <https://doi.org/10.1145/3582016.3582063>.
- [28] Paul E. McKenney. Hash datastructure. [https://github.com/paulmckrcu/perfbook/blob/master/CodeSamples/datastruct/hash/hash\\_bkt\\_rcu.c](https://github.com/paulmckrcu/perfbook/blob/master/CodeSamples/datastruct/hash/hash_bkt_rcu.c), October 2024. (Accessed on 10/28/2024).
- [29] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of The12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE ’16, page 25–35, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339476. doi: 10.1145/2892242.2892258. URL <https://doi.org/10.1145/2892242.2892258>.
- [30] Inc. Micron Technology. Micron launches memory expansion module portfolio to accelerate cxl 2.0 adoption. <https://investors.micron.com/news-releases/news-release-details/micron-launches-memory-expansion-module-portfolio-acc> 2022.
- [31] Alan Nair, Sandeep Kumar, Aravinda Prasad, Ying Huang, Andy Rudoff, and Sreenivas Subramoney. Telescope: Telemetry for gargantuan memory footprint applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 409–424, Santa Clara, CA, July 2024. USENIX Association. ISBN 978-1-939133-41-0. URL <https://www.usenix.org/conference/atc24/presentation/nair>.
- [32] Aleix Roca Nonell, Balazs Gerofi, Leonardo Bautista-Gomez, Dominique Martinet, Vicenç Beltran Querol, and Yutaka Ishikawa. On the applicability of pebs based online memory access tracking for heterogeneous memory management at scale. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, pages 50–57, 2018.
- [33] SeongJae Park. Memory access workload simulator. <https://github.com/sjp38/masim>, 2024. (Accessed on 10/28/2024).
- [34] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th Interna-*

- tional Middleware Conference Industrial Track*, Middleware '19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370417. doi: 10.1145/3366626.3368125. URL <https://doi.org/10.1145/3366626.3368125>.
- [35] Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John F. J. Mellor, Irina Higgins, Antonia Creswell, Nathan McAleese, Amy Wu, Erich Elsen, Siddhant M. Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, L. Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, N. K. Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Tobias Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d’Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew G. Johnson, Blake A. Hechtman, Laura Weidinger, Iason Gabriel, William S. Isaac, Edward Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem W. Ayoub, Jeff Stanway, L. L. Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. Scaling language models: Methods, analysis & insights from training gopher. *ArXiv*, abs/2112.11446, 2021.
- [36] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483550. URL <https://doi.org/10.1145/3477132.3483550>.
- [37] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Dae-woo Kim, and Dong Li. Mtm: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 803–817, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704376. doi: 10.1145/3627703.3650075. URL <https://doi.org/10.1145/3627703.3650075>.
- [38] Bala Narasimhan Robert Pang. Linux huge pages | google cloud blog, 8 2021. URL <https://cloud.google.com/blog/products/databases/cloud-sql-postgresql-now-supports-linux-huge-pages>. [Online; accessed 2025-03-16].
- [39] Steve Scargall. Using linux kernel memory tiering, 2022. URL <https://stevescargall.com/blog/2022/06/using-linux-kernel-memory-tiering/>. [Online; accessed 2025-03-18].
- [40] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. vtm: Tiered memory management for virtual machines. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 283–297, 2023.
- [41] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Anand Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. Using deep-speed and megatron to train megatron-turing nlq 530b, a large-scale generative language model. *ArXiv*, abs/2201.11990, 2022.
- [42] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 105–121, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614256. URL <https://doi.org/10.1145/3613424.3614256>.
- [43] tejasaks. Performance best practices for sql server on linux - sql server | microsoft learn, 11 2024. URL <https://learn.microsoft.com/en-us/sql/linux/sql-server-linux-performance-best-practices?view=sql-server-ver16>. [Online; accessed 2025-03-16].
- [44] ESX VMware. Performance evaluation of amd rvi hardware assist. 2008.
- [45] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS

'22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507731. URL <https://doi.org/10.1145/3503222.3507731>.

- [46] Wikipedia. Dynamic random-access memory. [https://en.wikipedia.org/wiki/Dynamic\\_random-access\\_memory](https://en.wikipedia.org/wiki/Dynamic_random-access_memory), Jun 2024. (Accessed on 06/23/2024).
- [47] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304024. URL <https://doi.org/10.1145/3297858.3304024>.
- [48] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. Parsec3. 0: A multicore benchmark suite with network stacks and splash-2x. *ACM SIGARCH Computer Architecture News*, 44(5):1–16, 2017.
- [49] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with cxl in virtualized environments. In *Symposium on Operating Systems Design and Implementation*, 2024.