Making Dynamic Page Coalescing Effective on Virtualized Clouds

Weiwei Jia*
The University of Rhode Island

Jiyuan Zhang*
New Jersey Institute of Technology

Jianchen Shan
Hofstra University

Xiaoning Ding
New Jersey Institute of Technology

Abstract

Using huge pages has become a mainstream method to reduce address translation overhead for big memory workloads in modern computer systems. To create huge pages, system software usually uses page coalescing methods to dynamically combine contiguous base pages. Though page coalescing methods help effectively reduce address translation overhead on native systems, as the paper shows, their effectiveness is substantially undermined on virtualized platforms.

The paper identifies this problem and analyzes the causes. It reveals and experimentally confirms that only huge guest pages backed by huge host pages can effectively reduce address translation overhead. Existing page coalescing methods only aim to increase huge pages at each layer, and fail to consider this cross-layer requirement on the alignmentment of huge pages.

To address this issue, the paper designs Gemini as a cross-layer solution that guides the formation and allocation of huge pages in the guest and the host. With Gemini, the memory management at one layer is aware of the huge pages at the other layer, and manages carefully the memory regions corresponding to these huge pages. This is to increase the potential of forming and allocating huge pages from these regions and minimize the associated cost. Then, it guides page coalescing and huge page allocation to first consider these regions before other memory regions. Because huge pages are preferentially formed and allocated from these regions and less from other regions, huge guest pages backed by huge host pages can be increased without aggravating the adverse effects incurred by excessive huge pages.

Extensive evaluation based on the prototype implementation in Linux/KVM and diverse real-world applications, such as key-value store, web server, and AI workloads, shows that Gemini can reduce TLB misses by up to 83% and improve application performance by up to 126%, compared to state-of-the-art page coalescing methods.

CCS Concepts: • **Networks → Cloud computing**; • **Software and its engineering**;

Keywords: Cloud Computing, Memory Management, Virtualization, Operating Systems

ACM Reference Format:
Weiwei Jia, Jiyuan Zhang, Jianchen Shan, and Xiaoning Ding. 2023. Making Dynamic Page Coalescing Effective on Virtualized Clouds. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3567487

1 Introduction

In modern computer systems, translation lookaside buffer (TLB) capacity cannot scale at the same rate as memory capacity [1, 2]. Address translation overhead has become a major performance bottleneck for many big-memory workloads [3–19]. This problem is particularly serious in virtualized clouds, where hardware supported nested paging is used to support memory virtualization (e.g., Intel's extended page tables [20] and AMD nested page tables [21]). With nested paging, to resolve a TLB miss, the processor needs to walk through two layers of page tables (i.e., two-dimensional page walk), and the cost can be 6x as much as walking through one layer of page table upon TLB misses on native systems [12, 20].

Using huge pages (e.g., 2MB pages on x86 platforms) to reduce address translation overhead has become a mainstream and effective method. With the huge page support in TLB, a TLB entry can cache the page table entry (PTE) of a huge page, and can be used to translate addresses for an increased amount of data (e.g., 2MB with a huge page PTE vs. 4KB with a base page PTE). This significantly increases TLB coverage, and thus reduces TLB misses. In addition, using huge pages can reduce the steps in a page walk and the memory reads incurred by the page walk.

To reduce address translation overhead with huge pages, system software needs to create and allocate huge pages.

*equal contribution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '23, May 8–12, 2023, Rome, Italy
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567487

Dynamic page coalescing (e.g., Linux transparent huge page) has become a de facto method for creating and allocating huge pages on many systems. Because huge pages may incur memory space waste and high demand-paging overhead, existing systems usually support simultaneously multiple page sizes (e.g., 2MB huge pages and 4KB base pages). They use page coalescing methods to dynamically combine contiguous base pages into huge pages to reduce address translation overhead and split under-utilized huge pages to reduce the space and paging overhead.

A common understanding is that the more huge pages are created and used in the system, the more effectively the address translation overhead can be reduced. Thus, many efforts are being focused on exploiting coalescing opportunities and reducing coalescing overhead, such that more huge pages can be created with low overhead [1, 5, 8, 15, 22].

Though this approach achieves great success in reducing address translation overhead on native (non-virtualized) systems, the paper shows that more huge pages may not necessarily mean "more effective" on virtualized platforms. As we will explain in §2, only when a huge page formed in the guest is backed by a huge page in the host, can the huge pages effectively reduce address translation overhead. We refer to them as *well-aligned huge pages* for brevity.

Though there are system components coalescing base pages into huge pages at both the guest and the host layers, because they coalesce pages independently, it is likely that a huge page formed in the guest is not backed by a huge page in the host, or vice versa. In these cases, the huge pages can hardly reduce address translation overhead. We refer to these huge pages as *mis-aligned huge pages* and refer to this problem as *huge page misalignment problem*. This problem can degrade performance by up to 67% as shown in §6.

To mitigate this huge page misalignment problem, the paper proposes Gemini as a cross-layer solution. Gemini guides and drives the huge page management in the guest and the host, particularly the page coalescing components, to turn mis-aligned huge pages into well-aligned huge pages by forming and allocating new huge pages to match the mis-aligned huge pages.

To achieve this goal, Gemini first makes the memory management at a layer aware of the mis-aligned huge pages formed at the other layer. Gemini periodically scans the process page tables in VMs and the VM page tables in the host to find the mis-aligned huge pages. It keeps track of these huge pages using their guest physical addresses. Thus, a guest can check the guest physical addresses of the mis-aligned huge pages in the host, and tries to form and allocate huge guest pages at these addresses. The host can check the guest physical addresses of the mis-aligned huge pages in a guest, and tries to back these addresses using huge host pages.

To form and allocate new huge pages that match the mis-aligned huge pages at the other layer, Gemini carefully manages the space of the memory regions corresponding to the mis-aligned huge pages. It reserves the space temporarily, hoping that the space can be allocated directly as huge pages or allocated as contiguous base pages, which later can be promoted into huge pages with minimal overhead. Then, Gemini guides page coalescing and huge page allocation to form and allocate huge pages from these regions first before considering other memory regions. These huge pages turn the mis-aligned huge pages at the other layer into well-aligned huge pages. Gemini does not create huge pages excessively, and thus does not aggravate the adverse effects incurred by excessive huge pages.

The paper makes the following contributions. First, to our best knowledge, this is the first work that identifies and studies the huge page misalignment problem in virtualized clouds. Second, we have proposed Gemini as an effective system solution that can efficiently address the problem and the technical challenges of the solution. Finally, we have implemented Gemini based on Linux and KVM. We also tested it and compared it to seven related systems, using diverse real-world applications and extensive experiments. Our tests show Gemini can significantly reduce TLB misses and address translation overhead, and effectively improve application performance and system efficiency compared to other evaluated systems.

2 Background and Motivation

This section first introduces how virtual addresses are translated with TLB and what the overhead is (§2.1). Based on the introduction, it explains how huge pages help reduce the overhead on native systems and why they may not reduce it on virtualized systems, i.e., the huge page misalignment problem (§2.2). Then, it explains and experimentally confirms that, due to the huge page misalignment problem, dynamic page coalescing becomes ineffective on virtualized platforms (§2.3).

2.1 Address Translation with TLB and Its Overhead

To access any data in CPU cache or memory, virtual memory addresses must be translated to the real memory locations of the data, i.e., physical memory addresses on a native system or host physical address (HPAs) on a virtualized system. Existing systems use predominantly page-based memory management. The address mappings needed to finish the translations are managed in page tables. On a native system, process page tables in the OS manage the mappings between virtual pages and physical pages. A virtualized system usually uses nested paging. The process page tables in the guest OS manage the mappings between guest virtual pages (GVPs) and guest physical pages (GPPs), and the virtual machine (VM) page table in the host OS manage the mappings between GPPs and host physical pages (HPPs). The mappings in these two layers of page tables must be combined to form complete translations.

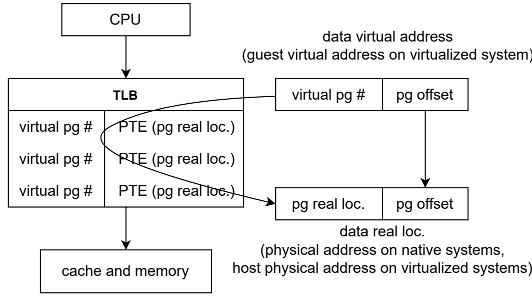

Figure 1. Address translation with a TLB on native systems and on virtualized systems.

The address translation overhead mainly refers to the time required to finish address translations, i.e., address translation latencies. In a processor, the TLB conducts address translations, and plays an importantly role in reducing the overhead. A TLB is a small cache that buffers a number of page table entries (PTEs), which contain the real memory locations of the corresponding virtual pages. On a native system the PTEs of process page tables are cached; and on a virtualized system the PTEs of VM page tables are cached. The PTEs are tagged with the virtual page addresses; this forms a table mapping virtual pages to their real locations. As shown in Figure 1, to translate a virtual address, the TLB uses the virtual page address (higher bits in the virtual address) to find the corresponding page table entry and obtain the real location of the page. Then, it adds the page offset (lower bits in the virtual address) to the page location to obtain the location of the data.

When the PTE needed to finish an address translation can be found in the TLB (i.e., a TLB hit), the address translation overhead is minimized. Otherwise (i.e., a TLB miss), the TLB conducts a page walk to locate the PTE and load it to the TLB. On a native system, this is to walk down a multi-level process page table. On a virtualized system, it is much more complex. The page walk is essentially two-dimensional, with one dimension being walking through the process page table in the guest and the other dimension being walking through the VM page table in the host. Interested readers can refer to [12] for details.

TLB misses and page walks can significantly increase address translation overhead, because they may incur memory accesses. For example, on x86 platforms, each page walk may incur up to 4 memory accesses on a native system and 24 memory accesses on a virtualized system with nested paging. In addition to memory accesses, in two-dimensional page walks, since the guest physical addresses used in process page tables in the guest must be translated to host physical addresses, extra TLB misses may be incurred, further increasing the address translation overhead.

To reduce the overhead incurred by page walks, page walk caches are integrated in TLB. They cache the page table directories that must be accessed and used to locate PTEs. Thus, accessing these page table directories can be satisfied in TLB without reaching to memory. Page walk caches are particularly effective in caching high-level page table directories that are close to the tree root of a page table [12]. However, it is not easy to cache the directories at the lowest level of page tables that are close to the PTEs of base pages [23]. Thus, TLB misses are still much more costly than TLB hits; and reducing TLB misses continues to be a paramount task.

2.2 Huge Page Misalignment Problem

Huge pages may reduce address translation overhead in two ways.

• Reducing TLB misses, as the primary way: When cached in the TLB, the PTE of a huge page can be used to translate addresses for hundreds times more data than the PTE of a base page (2MB vs. 4KB). Thus, using huge pages significantly increases the coverage of a TLB, and thus reduces TLB misses.

• Reducing the overhead of page walks, as a secondary way: The PTE of a huge page is closer to the root of the page table. Thus, in a page walk, it takes fewer step(s) to walk down the page table to locate the PTE of a huge page than the PTE of a base page. More importantly, only high-level page table directories are needed to locate the PTEs of huge pages, which can easily be cached in page walk caches. This makes the page walk overhead substantially lower for huge pages than base pages.

On a native system, when accessing the data in any huge pages, the PTEs of the huge pages can be cached in TLB to reduce TLB misses. Thus, the more huge pages are created and used, the more address translation overhead can be reduced. However, on a virtualized system, when accessing the data in a huge page, as explained below, it is likely that there is not a PTE can be cached in the TLB. Thus, rather than reducing TLB misses, using huge pages may even increase TLB misses.

Using huge pages can reduce TLB misses only when both the guest and the host use huge pages for the same data, i.e., a huge GVP is backed by a huge GPP and a huge HPP at the same time. The reason is that only in these cases the PTEs of the VM page tables can be cached in the TLB and used correctly in address translation [12]. When a huge GVP is backed by multiple base HPPs in the host, there is not a PTE in the VM page table that corresponds to the GVP; thus, no PTEs can be loaded to the TLB to help translation. If a base GVP is backed by a huge HPP, because there is not a PTE in the VM page table that corresponds to the virtual page, the page offset cannot be used to obtain correct host physical address (HPA). Thus, in neither of these cases, there is a valid PTE that can be cached in the TLB to help the address translation. The paper refers to such huge pages as *misaligned huge pages* and this problem as *huge page misalignment problem.* For brevity, we refer to the huge pages

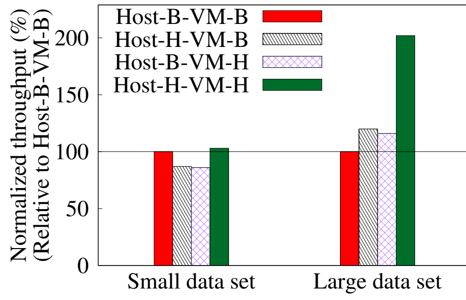

Figure 2. Misaligned huge pages cannot reduce address translation overhead.

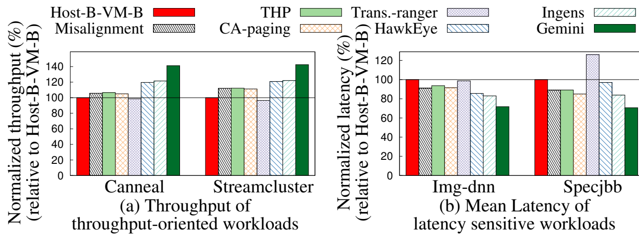

Figure 3. *Huge page misalignment problem in virtualized systems.* Compared to existing systems, Gemini's performance is much better because most huge pages are well aligned between guest- and host-level.

backed or being backed by huge pages as *well-aligned huge pages*.

Though the misaligned huge pages still can help reduce page walk overhead, they increase TLB misses. Thus, they can hardly reduce address translation overhead, and may even hurt performance. To illustrate this problem, we show in Figure 2 the performance of a micro-benchmark in a VM that randomly access a data set. When the data set is small, no TLB misses are incurred, and the performance is similar, no matter whether base pages or huge pages (aligned) are used in the guest and the host to save the data set (labeled with *Host-B-VM-B* and *Host-H-VM-H*, correspondingly, in the figure). However, if huge pages are only used in one layer, and base pages are used in the other layer (labeled with *Host-B-VM-H* and *Host-H-VM-B* in the figure), these huge pages become misaligned huge pages. Accesses to the data in these pages cause TLB misses, and makes the microbenchmarks show lower performance. When the data set is large, using well-aligned huge pages can substantially improve performance, since they can reduce TLB misses and page walk overhead at the same time. Using misaligned huge pages can hardly improve performance, compared to using only base pages, since the benefits of reducing page walk overhead are largely offset by increased misses.

2.3 Page Coalescing Efforts Invested in Vain

On a virtualized platform, the guest and the host may independently coalesce contiguous base pages into huge pages

workloads	THP	CA-paging	Trans.-ranger	HawkEye	Ingens	Gemini
Canneal	26%	32%	23%	29%	30%	51%
Streamcluster	19%	22%	15%	25%	35%	50%
Img-dnn	21%	18%	15%	33%	35%	67%
Specjbb	18%	14%	12%	32%	33%	81%

Table 1. *Rates of well-aligned huge pages.* Existing systems cannot effectively manage huge pages in virtualized system because rates of well-aligned huge pages in these systems are low, greatly increasing TLB misses and the address translation overhead.

or split huge pages into base pages. Due to the lack of coordination, the huge pages are "well-aligned" largely by chance. Though the chance increases when more huge pages are created in each layer, the pressure to reduce the adverse effects of huge pages (e.g., space waste and paging overhead) caps the chance, and thus limits the effectiveness of page coalescing mechanisms.

To understand how this problem affects the performance of different types of workloads, we test the performance of two throughput-oriented applications from `PARSEC` benchmark suite [24], `Canneal` and `Streamcluster`, and two latency-sensitive applications from `TailBench` [25], `Img-dnn` and `Specjbb`. §6 details the experimental setup, including server/VM configurations and application descriptions. We compare the performance of these applications (Figure 3) and the rates of well-aligned huge pages (Table 1) under four scenarios.

- **Base page only**: Both the guest and the host use only base pages. This scenario is labeled as `Host-B-VM-B` in Figure 3.
- **All huge pages are mis-aligned**. The guest only allocates base pages to the application, and the host only allocates huge pages (labeled with `Misalignment`)[1].
- **Uncoordinated page coalescing** is used in both host and guest to create huge pages. Thus only a small portion of huge pages may be well-aligned. We tested the latest page coalescing mechanisms, including `Ingens` [1], `HawkEye` [8], `CA-paging` [26], `Translation-ranger` [27], as well as Linux `THP` [28].
- Our solution **Gemini**, which can make most huge pages well-aligned.

The results first show that, compared to the base-page-only scenario, using huge pages only incrementally improve performance if they are mis-aligned (`Misalignment`). This is consistent with what we have observed with the micro-benchmark earlier. Using `THP`, `CA-paging`, or `Translation-ranger` in both the host and the guest can create some well-aligned huge pages (below 30%). However, the reduced address translation overhead can hardly be reflected on performance. Compared to the `Misalignment` scenario, the applications even show lower performance

[1]When the guest allocates huge pages and the host allocates base pages, the results are similar.

with Translation-ranger, mainly because of the high run-time overhead paid on page promotion operations. Ingens and HawkEye can create more huge pages with low overhead. Thus, the rates of well-aligned huge pages increase to around 30%, are reflected by non-trivial performance improvements. However, run-time overhead and other system overhead (e.g., space waste) cannot allow further increasing well-aligned huge pages through increasing the total number of huge pages. Finally, the results show that, with Gemini, more than 50% of huge pages are well-aligned. This is achieved without increasing the total number of huge pages or high system/run-time overhead. Thus, it improves the throughputs by over 20% and lower the latencies by over 16% on average, relative to Ingens and HawkEye. We also collected TLB misses in the experiments, which show similar trend and are included in §6.

3 Gemini: Objective and Main Idea

As explained and confirmed in §2.3, only well-aligned huge pages can effectively reduce address translation overhead. Thus, the objective of Gemini is to effectively turn mis-aligned huge pages into well-aligned huge pages. The method is to create new huge pages and map them to mis-aligned huge pages.

Gemini classifies mis-aligned huge pages into two types. To turn them into well-aligned huge pages, Gemini tries to use different methods to create new huge pages. A *type-1 mis-aligned huge page* does not have any base pages mapped to it. To turn it into a well-aligned huge page, Gemini tries to allocate a huge page at the corresponding address at the other layer. If the opportunity of allocating such a huge page is not immediately available, Gemini tries to allocate contiguous base pages, which later can be directly promoted into a huge page without any page migration.

A type-1 mis-aligned huge page turns into a *type-2 mis-aligned huge page*, if the above two methods fail. A type-2 mis-aligned huge page has some base pages mapped to it; but the base pages cannot be directly promoted into a huge page, for which page migrations must be involved. To turn a type-2 mis-aligned huge page into a well-aligned huge page, Gemini tries to use the existing system component for page coalescing to promote the corresponding base pages into a huge page.

Note that we are seeking a solution with low overhead. Creating excessive huge pages causes both space overhead (e.g., memory fragmentation) and run-time overhead (e.g., page migrations). The overhead constraint has been the main reason for existing systems to use multiple page sizes and dynamic page coalescing methods. It is also a main factor affecting the design of Gemini.

To reduce the overhead while keeping the effectiveness in turning mis-aligned huge pages into well-aligned huge pages, Gemini uses three techniques. First, Gemini temporary maintains the status of type-1 mis-aligned huge pages until they becoming well-aligned huge pages or time-out. Compared to type-2 mis-aligned huge pages, type-1 mis-aligned huge pages incur much lower space overhead and run-time overhead when turned into well-aligned huge pages. Maintaining the status of type-1 is to prevent them from becoming type-2 mis-aligned huge pages and thus avoid the high overhead associated with turning type-2 mis-aligned huge pages. The status maintaining is achieved by temporarily reserving the memory regions corresponding to the type-1 mis-aligned huge pages. During the reservation, only the allocations of huge pages and the allocations of contiguous base pages are allowed.

Second, Gemini enhances page allocators. With the enhancement, when huge pages or contiguous base pages are needed, the memory regions reserved for maintaining the status of type-1 mis-aligned huge pages are used first in the allocations. This is also to increase the chances that turn type-1 mis-aligned huge pages directly to well-aligned huge pages without going through the type-2 status.

Third, when page coalescing components are used to promote pages, they first try to promote the base pages mapped to type-2 mis-aligned huge pages before checking other base pages. This is to increase the chance that type-2 mis-aligned huge pages are turned into well-aligned huge pages and to leverage the mechanisms in these page coalescing components to avoid high overhead and excessive page promotions.

4 Gemini: Overall Structure and Key Techniques

Figure 4 shows the key components in Gemini and other related system components that work together to implement the idea introduced in Section 3.

To fix mis-aligned huge pages, the first step is to detect them. This is conducted using the host layer misaligned huge page scanner (MHPS) component as shown in Figure 4. MHPS periodically scans the page tables of the guest processes for the huge pages formed in the guest, and scans the page tables of VMs for the huge pages formed in the host. For each huge page identified in the scanning, MHPS labels it with the system layer (i.e. guest or host), its guest physical address, and VM ID of the address. Mis-aligned huge pages can be identified by comparing the labels.

Then, Gemini makes each guest aware of the mis-aligned huge host pages that are mapped to it. This is achieved by providing the guest physical addresses of the mis-aligned host huge pages labeled with the corresponding VM ID. The guest can check the pages allocated to these addresses to determine the types (type-1 or type-2) of these mis-aligned huge pages.

At each layer, Gemini uses the huge booking component to reserve the huge-page-sized memory space corresponding to the type-1 mis-aligned huge pages. Such a memory region is reserved until a time-out is reached or until the region is allocated as a huge page or contiguous base pages. The

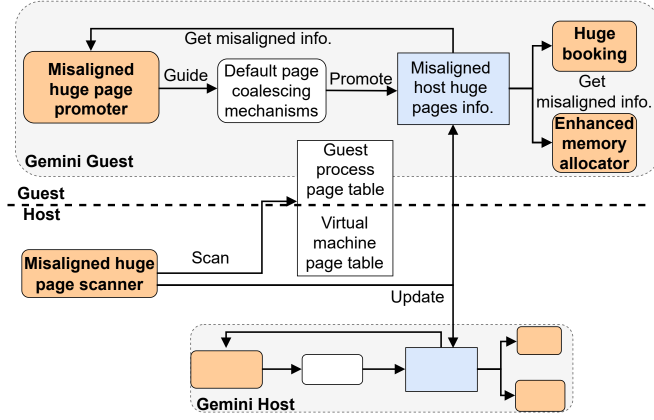

Figure 4. *Gemini system overview.* Key components are shaded in orange.

Algorithm 1 Booking Timeout Adjustment

1: T_d: desired timeout value; T_e: effective timeout value; T_{init}: initial timeout value; P: time period between two adjustments
2: $T_d \leftarrow T_{init}$
3: **while** true **do**
4: $T_e \leftarrow T_d$, collect TLB misses and memory fragmentation for a time period of P
5: **if** TestTimeout(T_d * 1.1) **then**
6: $T_d \leftarrow T_d * 1.1$; continue
7: **else**
8: $T_e \leftarrow T_d$, collect TLB misses and memory fragmentation for a time period of P
9: **end if**
10: **if** TestTimeout(T_d * 0.9) **then**
11: $T_d \leftarrow T_d * 0.9$; continue
12: **end if**
13: **end while**
14: **function** TestTimeout(T)
15: $T_e \leftarrow T$, collect TLB misses and memory fragmentation for a time period of P
16: $D_{TLB} \leftarrow$ average decrease of TLB misses
17: $D_{Frag} \leftarrow$ average decrease of memory fragmentation
18: **if** $D_{TLB} > 0$ and $D_{Frag} >= 0$ **then return** true; **end if**
19: **return** false
20: **end function**

method for adjusting the timeout value will be introduced in Section 4.1.

The enhanced memory allocator interacts with the huge booking component to preferentially allocate huge pages and contiguous base pages from the memory regions reserved by the huge booking component. We introduce how base pages are allocated from these regions in Section 4.2.

In addition to these components, the promoter component interacts with the page coalescing component in the system to perferentially promote the base pages mapped to type-2 mis-aligned huge pages.

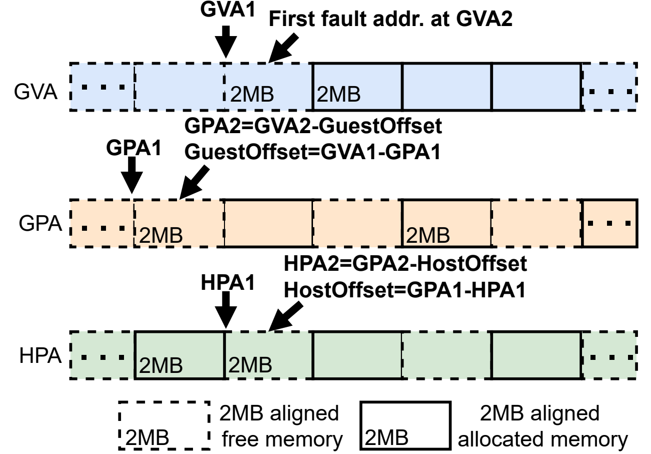

Figure 5. *Main idea of EMA.* Each huge page sized memory region is associated with a starting address and an offset in each level. The starting address (e.g., GVA1) of one huge page sized memory region is the starting address of the huge page that the first fault address (e.g., GVA2) belongs to. To form well-aligned huge pages, "$GuestOffset = GVA_{start} - GPA_{start}$" and "$HostOffset = GPA_{start} - HPA_{start}$" are used for forthcoming memory allocations in guest- and host-level, respectively.

4.1 Booking Timeout Value Adjustment

This *booking timeout* value is a key parameter in Gemini. A large value increases the waste of memory space and may increase memory fragmentation, as the memory space cannot be used by other processes or for other purposes during the booking period. A small value may reduce the effectiveness of Gemini in reducing its overhead. Gemini uses Algorithm 1 to periodically adjust the time period to improve Gemini's effectiveness without increasing memory fragmentation. The algorithm slightly increases or decreases the booking timeout value, and checks how TLB misses and the degree of memory fragmentation changes. It keeps the new value if TLB misses are decreased and the degree of memory fragmentation is not increased. It uses Linux `perf` tool to measure the number of TLB misses, and uses free memory fragmentation index (FMFI [1]) to measure the degree of memory fragmentation.

4.2 Enhanced Memory Allocator (EMA)

Gemini enhances the memory allocator to form huge pages from the memory regions reserved by the huge booking component. For example, for a 2MB type-1 mis-aligned huge page in the host, based on the guest physical address of the page, the huge booking component in the guest reserves a 2MB guest physical memory region. If a huge page needs to be allocated in the guest, the enhanced memory allocator (EMA) can allocate this memory region to fit the need. This turns the mis-aligned huge page into a well-aligned huge page. However, often such an opportunity is not available, and only requests for base pages are available. In this case, EMA tries to allocate the space in this region in the form of contiguous base pages and later directly promote these base pages into a huge page. This requires that these base pages

have contiguous guest virtual addresses, and the lowest address is aligned to the huge page size (2MB). This subsection mainly introduces how this requirement can be satisfied.

The main idea of EMA is to align the starting addresses of GPA and HPA to GVA based on huge pages upon the first page fault to the huge page sized memory region. For instance, in Figure 5, upon the first page fault at GVA2, Gemini allocates guest physical memory space starting at GPA2. GPA2 is aligned to GVA2 based on huge pages.

To achieve this, Gemini first locates the starting address of the huge page sized guest virtual memory space (GVA1 in Figure 5) that the first fault address (GVA2) belongs to. Then, Gemini finds a free huge page sized guest physical memory space. Please note that Gemini first chooses the guest physical memory space that has been backed by host huge pages (i.e., misaligned host huge pages). The misaligned huge pages information is collected and shared to the guest by the host level misaligned huge page scanner (MHPS) as shown in Figure 4. Next, Gemini locates the starting address of the huge page sized guest physical memory region (GPA1) and calculates the offset of the huge page sized memory region for the guest level (i.e., $GuestOffset = GVA1 - GPA1$). Finally, Gemini calculates where to allocate guest physical memory space (i.e., $GPA2 = GVA2 - GuestOffset$).

If the guest physical space is not backed by host physical space, the starting address of HPA is fault at HPA2 that is aligned to GPA2 based on huge pages. The offset in the host level is calculated in the same way as that in the guest level (i.e., $HostOffset = GPA1 - HPA1$). For forthcoming memory allocations, $GuestOffset$ is used as the guest level offset to calculate where to allocate GPA; $HostOffset$ is used as the host level offset to calculate where to allocate HPA. By doing so, in-place huge page promotion can be used to increase the efficiency of forming well-aligned huge pages.

Our key observation is that the number of well-aligned huge pages can be significantly increased with low overhead, if memory space corresponding to the misaligned huge page is allocated based on the above huge booking and EMA mechanisms.

Huge preallocation. The above requirement (512 base pages with contiguous guest virtual addresses and the lowest address being aligned to the huge page size) may not be met easily. In many cases, only a few base pages are missed to make up to such 512 pages in the requirement. In these cases, EMA pre-allocates the missing base pages, so as to promote the base pages early. EMA performs the pre-allocation and the promotion before the huge booking time-out. To control the space waste, EMA performs the pre-allocation cautiously when two conditions are met: 1) the number of base pages allocated in the region exceeds a threshold (256 selected experimentally), and 2) the degree of memory fragmentation is low (FMFI≤0.5), indicating that the workload tends to use contiguous memory pages.

5 Implementation Details

We have implemented Gemini based on Linux/KVM 4.19. We added/modified about 5500 LoC mainly in Linux memory management subsystem. Specifically, these additions and modifications include: 1) ~1700 LoC in *page_alloc.c*, ~500 LoC in *huge_memory.c*, and ~400 LoC in *mempolicy.c* to implement the EMA and HB (huge booking) mechanisms; 2) ~2700 LoC in a new kernel file (*kgeminid.c*) to implement the misaligned huge page promoter (MHPP) mechanisms; 3) other user-level control programs to help users use Gemini.

Enhanced Memory Allocator. We realize EMA based on virtual memory areas (VMAs) instead of huge page sized memory regions. The reason is that the number of *offset descriptor* for huge page sized memory regions can be huge. It is hard to efficiently manage and search them. When Gemini memory allocations are requested, EMA first tries to find the *offset descriptor* associated with the VMA that the requested address is in. The *offset descriptor* includes the offsets to calculate where to allocate the guest/host physical address. The *offset descriptors* are organized in a self-organizing linear search list [29] to optimize the search time. If the *offset descriptor* of the VMA cannot be found, Gemini memory allocator calculates the offsets for the memory region, creates a new *offset descriptor*, and inserts it into the list.

To realize EMA based on VMAs, there are two main practical issues to overcome. First, what guest/host physical memory space should be booked to fit the entire VMA. Gemini tries to align the entire VMA based on huge pages boundaries instead of aligning each fault address individually. This is because fitting the entire VMA can increase memory contiguity and reduce memory fragmentation [1, 27]. Many previous works [2, 26] also show that memory contiguity can significantly reduce address translation overhead if TLB is enhanced to support larger memory (> 1GB) translation in the future.

To address the first issue, Gemini enhances the buddy allocator in Linux. Existing memory allocator usually uses binary buddy allocator [30, 31] as it is faster compared to other allocators [32]. In buddy allocator (e.g., Linux buddy system), free memory is grouped into order-x free memory blocks lists, where a memory block in an order-x free list incorporates 2^x contiguous free pages. Typically, the maximum order is 11 (e.g., Linux MAX_ORDER attribute); thus, existing buddy allocator can only allocate up to 4MB contiguous memory space (i.e., order-11). Further increasing the order number is very sensitive to external fragmentation [26, 33]. Therefore, existing buddy allocator cannot be directly used for EMA.

Second, how to handle the target GPA or HPA is unavailable for allocation. After the guest/host memory space is booked, the target GPA/HPA calculated with aforementioned EMA mechanisms may not be available for allocation as the

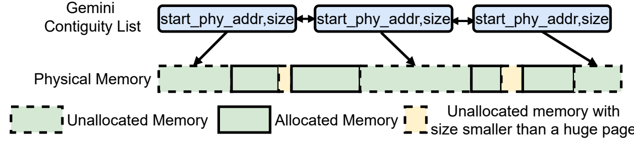

Figure 6. *Gemini contiguity list.*

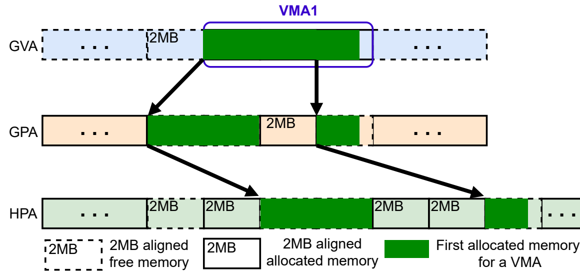

Figure 7. *Sub-VMA.*

corresponding VMA may be changed (e.g., VMA expansion [34]) or the target GPA/HPA has been allocated. Gemini should tolerate such issue. Gemini realizes Gemini *contiguity list* and Sub-VMA to address the above two issues.

Gemini *contiguity list.* Figure 6 shows the Gemini *contiguity list*. It tracks the contiguous physical memory regions. Each entry in *Gemini contiguity list* points to a free and contiguous physical memory region with its starting physical address and size. Gemini is used when the size of the VMA is larger than the huge page size (i.e., 2MB). *Gemini contiguity list* is sorted based on the starting address to mitigate memory fragmentations. The reason is that small and random page allocations are allocated from the beginning part of the physical memory space without fragmenting large, free, and contiguous memory regions. When a VMA is touched for the first time, Gemini searches the *Gemini contiguity list* for a free physical memory region that could fit it with the next-fit policy. In the host level, Gemini searches the Gemini *contiguity list* of the HPA to fit the contiguous guest physical memory space that has been firstly touched. This increases the possibility of forming more well-aligned huge pages. The search starts from the place where it left off the previous time. Please note that Gemini only allocates the physical pages that are touched in the selected memory region; it does not allocate the entire physical memory region for the whole VMA; it directs the forthcoming page faults of the same source VMA to the selected free memory region through the aforementioned offset mechanisms. In case there is no available free memory regions to fit the entire VMA, Gemini finds the largest free physical memory region in the *Gemini contiguity list*. Gemini uses the sub-VMA mechanisms to process the remaining VMA. Currently, we realize the contiguity list with double link list in Linux.

Sub-VMA. Figure 7 shows the sub-VMA mechanism. After Gemini searches Gemini *contiguity list*, it may not find an available memory region to fit the entire VMA such as VMA1 in Figure 7. In this case, Gemini chooses the largest free memory region and generates a new starting address, a new length, and a new offset for the remaining VMA. For the forthcoming memory allocations, Gemini locates the offset associated with the VMA through checking each VMA's starting address and size. If the VMA is found, Gemini calculates the target GPA and HPA with the corresponding offset; if the target GPA or HPA is unavailable for allocation (e.g., already allocated), Gemini chooses a new free memory region for the remaining VMA from Gemini contiguity list. If no VMA is found (i.e., first touched VMA), Gemini searches Gemini *contiguity list* to find a free memory space for the first touched VMA. The sub-VMA mechanisms are applied to each level independently to avoid costly interactions between guest- and host-level.

Huge Bucket. We implement the huge bucket through repurposing the buddy allocator. Huge bucket books the misaligned host huge pages and each time allocates the whole huge page sized guest physical memory regions backed by host huge pages. The buddy allocator groups free memory pages into blocks and each block contains 2^x contiguous free pages and is aligned to $2^x \times 4KB$, where the non-negative x is the order of the block. We leverage order 9 and above to realize the huge memory bucket. For the well-aligned huge pages that have been freed after use, Gemini temporarily reserves them for a time period and returns them to the OSs afterwards. Gemini also returns some well-aligned huge pages when the memory space becomes scarce or the memory fragmentation becomes severe.

6 Evaluation

We have evaluated Gemini extensively with a diverse set of workloads and compared Gemini to seven related systems. The objective of the evaluation is four-fold: 1) to show that Gemini can improve performance with high efficiency when the workload runs in a clean slate VM (§6.2) and in an reused VM (§6.3), respectively, 2) to understand Gemini's performance advantage compared to the related systems (§6.2, §6.3, and §6.5), 3) to verify the effectiveness of the major techniques in Gemini (§6.4), and 4) to evaluate the applicability and overhead of Gemini (§6.5).

6.1 Experiment Settings

Our evaluation was conducted on a DELL™ PowerEdge™ T630 server with two 2.1GHz Intel Xeon E5-2620 processors, 128GB of DRAM, a 1.6TB SSD, and an Intel I350 Gigabit NIC. Each processor has 8 physical cores. Each core has 1536 L2 TLB entries for 4KiB/2MiB pages, 4 entries of data TLB for 1GiB pages, 64 entries of data TLB for 4KiB pages, 8 entries of instruction TLB for 2MiB/4MiB pages, and 64 entries of instruction TLB for 4KiB pages. With Linux QEMU/KVM, we built VMs, each with a single virtual CPU (vCPU) or multiple

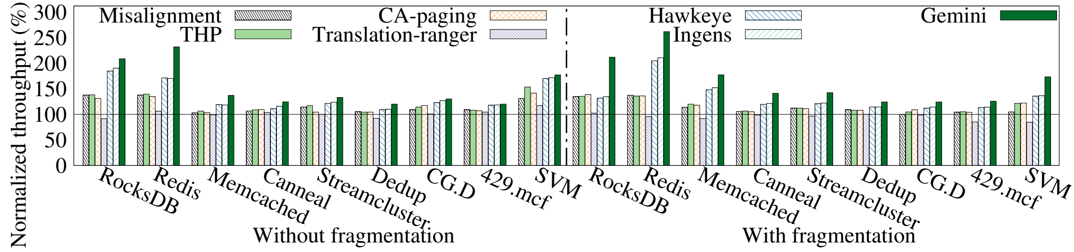

Figure 8. *Throughputs of different systems when workload runs in a clean slate VM.* Throughputs are normalized to those of `Host-B-VM-B`.

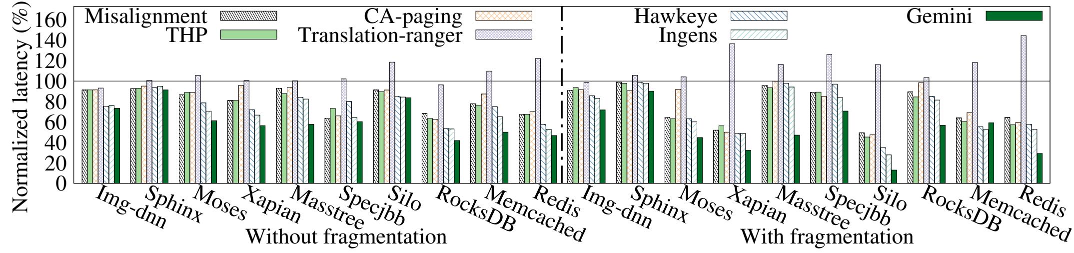

Figure 9. *Mean latencies of different systems when workload runs in a clean slate VM.* Mean latencies are normalized to those of `Host-B-VM-B`.

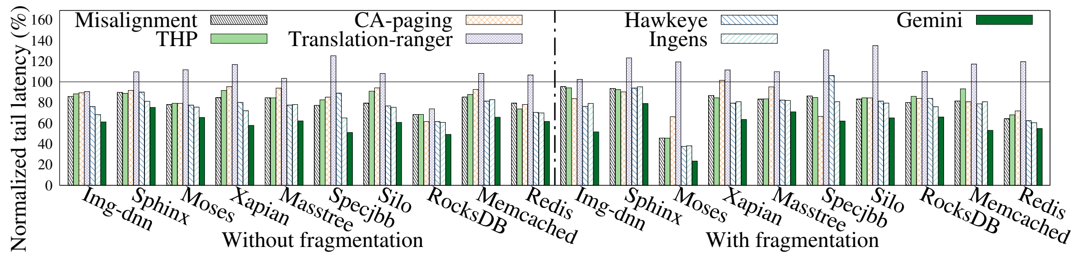

Figure 10. *Tail latencies (99th) of different systems when workload runs in a clean slate VM.* Tail latencies are normalized to those of `Host-B-VM-B`.

App.	Workload description
`Img-dnn`	Handwriting recognition based on OpenCV [35].
`Sphinx`	Speech recognition like Apple Siri [36].
`Moses`	Real time translation like Google translate [37].
`Xapian`	Search engine used in websites and S/W frameworks [38].
`Masstree`	In memory K/V store with 50% GET and 50% PUT [39].
`Specjbb`	Industry-standard JAVA middleware benchmark [40].
`Silo`	In-memory transactional database with TPCC [41].
`Shore`	On-disk transactional database with TPCC [42].
`RocksDB`	Serve requests (random keys,50% SET,50% GET) [43].
`Redis`	Serve requests (random keys,50% SET,50% GET) [44].
`Memcached`	Serve requests (random keys,50% SET,50% GET) [45].
`PARSEC`	Three benchmarks from `PARSEC` benchmark suite [24].
`NPB`	Two benchmarks from `NPB` benchmark suite [46].
`429.mcf`	Scientific computation benchmark in SPEC CPU 2006 [47].
`SVM`	The Benchmark for support vector machines [48].

Table 2. Programs and workloads used to test Gemini.

vCPUs and 32GB memory. We set the number of application threads equal to the number of vCPUs. Both host OS and guest OS are Ubuntu Linux 16.04 with the same Linux 4.19 kernel and software configuration, unless otherwise indicated. We test Gemini with a large and diverse set of workloads generated by typical applications from different domains (e.g., web server, database server, key/value store, AI workload, scientific applications, etc.), as summarized in Table 2. We profile these workloads with hardware performance counters, which show they all spend a significant part of execution time on page walks. This is also well corroborated by previous works [1, 8, 16, 26]. Two workloads (i.e., `Shore` and `SP.D`) are non-TLB sensitive and used to test the applicability and overhead of Gemini. In the experiments, each VM encapsulates one workload.

We test Gemini under two settings. Under the first setting, workload is executed in a clean slate VM. This is used to test the effectiveness of Gemini on forming well-aligned huge pages. In this setting, we test the workload with memory fragmentation and without memory fragmentation, respectively. We developed a program to fragment the memory. The program used the free memory fragmentation index (FMFI) to measure memory fragmentation [1, 8, 15]. In the experiment, both guest- and host-level memory are fragmented, unless otherwise indicated. We care more about Gemini's performance when memory is fragmented because previous works

show that memory quickly fragments in multi-tenant virtualized cloud environments [1, 49]. We further study the effectiveness of Gemini through comparing it to seven related systems. These systems include `Host-B-VM-B`, `Misalignment`, `THP` [28], `Ingens` [1], `HawkEye` [8], `CA-paging`[2] [26], and `Translation-ranger` [27]. Please refer to §2.3 for detailed descriptions about these systems.

Under the second setting, workload is executed in an reused VM. In this setting, we execute a workload after the completion of another workload (i.e., `SVM` [48]) with a large working set (~30GB). In virtualized systems, memory space allocated to the VM will not return to the host OS immediately [26, 51]. In this case, we want to test the effectiveness of Gemini's huge bucket mechanisms. Specifically, we want to evaluate whether the well-aligned huge pages can be efficiently reused by the workload.

We measure the throughputs of the workloads. We also collect average and tail latencies if the workloads report them. The performance measurements may vary significantly across different workloads. When we present them in figures, for clarity, we normalize them against those of `Host-B-VM-B` or Gemini, as indicated in the figures.

6.2 Workload Performance in A Clean Slate VM

Figure 8 shows throughputs of different workloads when all the eight systems are tested with memory fragmentation and without memory fragmentation, respectively. Gemini outperforms `Host-B-VM-B` by 1.72x on average, which is the best throughput among all the systems. To pinpoint why Gemini shows the best throughput, we profile the rates of well-aligned huge pages when workload is executed in each evaluated system. We show the results in Table 3. Gemini forms the largest rates of well-aligned huge pages, 66% on average. This helps Gemini reduce TLB misses significantly, as shown in Figure 11. The average rates of well-aligned huge pages in other systems are up to 33%, as these systems only consider increasing the efficiency of huge page management in each level. Gemini considers not only the efficiency of using huge pages in each level but also (more importantly) the effectiveness of forming well-aligned huge pages in virtualized systems.

Among all the systems, only `Translation-ranger` decreases the throughput by 7% on average, compared to `Host-B-VM-B`. The main reason is that page migrations incurred by `Translation-ranger` cause significant overhead. Specifically, `Translation-ranger` frequently migrates pages to increase memory contiguity, in order to increase TLB coverage. In comparison to `Host-B-VM-B`, all the systems other than Gemini and `Translation-ranger` improve the throughput by 22% on average, as these systems form either huge pages only in one level (e.g., `Misalignment`) or well-aligned huge pages by chance (e.g., `Ingens`, `HawkEye`, `CA-paging`, and `THP`).

workloads	THP	CA-paging	Trans.-ranger	HawkEye	Ingens	Gemini
Img-dnn	21%	18%	15%	33%	35%	67%
Sphinx	17%	17%	12%	31%	34%	64%
Moses	19%	16%	14%	26%	24%	57%
Xapian	18%	17%	13%	30%	34%	61%
Masstree	22%	16%	15%	33%	43%	70%
Specjbb	18%	14%	12%	32%	33%	81%
Silo	12%	16%	14%	23%	24%	62%
RocksDB	25%	17%	13%	33%	26%	64%
Redis	18%	17%	13%	35%	28%	60%
Memcached	17%	15%	12%	23%	25%	77%
Canneal	26%	32%	23%	29%	30%	51%
Streamcluster	19%	22%	15%	25%	35%	50%
dedup	18%	22%	15%	25%	26%	55%
CG.D	20%	23%	11%	42%	46%	80%
429.mcf	25%	27%	13%	36%	42%	76%
SVM	23%	20%	16%	41%	40%	81%

Table 3. Rates of well-aligned huge pages in different systems when workload runs in a clean slate VM.

Figure 9 shows the mean latency of all the systems when memory is fragmented and unfragmented, respectively. Gemini reduces the mean latency by 57% on average, compared to `Host-B-VM-B`. In comparison to `Host-B-VM-B`, `Translation-ranger` increases the mean latency by 11% on average, as page migrations incurred by `Translation-ranger` frequently trigger costly TLB shootdowns and CPU caches pollution. This significantly increases the mean latency in virtualized systems [52–54]. Nevertheless, page migrations also opportunistically increase the rate of well-aligned huge pages. That's why `Translation-ranger` forms 14% more rate of well-aligned huge pages and incurs 27% lower TLB misses on average, relative to `Host-B-VM-B`.

Figure 9 also shows that all the systems other than Gemini and `Translation-ranger` reduce the mean latency by 24% on average, compared to `Host-B-VM-B`. `Misalignment` can reduce the mean latency as it forms huge pages in the host OS. For `Ingens`, `HawkEye`, `CA-paging`, and `THP`, they form well-aligned huge pages by chance, such that they may reduce TLB misses and address translation overhead. Interestingly, for workload `Specjbb`, `HawkEye` increases the mean latency and the tail latency by 15% and 26% on average, respectively, compared to `Ingens`. We profile `HawkEye` and find that it deduplicates `Specjbb`'s in-use zero-pages and incurs extra copy-on-write page faults. This leads to the higher latency compared to `Ingens`.

Figure 10 shows the 99th tail latencies of different systems when memory is fragmented and unfragmented, respectively. Compared to `Host-B-VM-B`, Gemini and other systems reduce the tail latency by 60% and 14% on average, respectively. Gemini reduces much more tail latency for two main reasons. First, other systems do not consider the design

[2] We tested `CA-paging`'s software component [50].

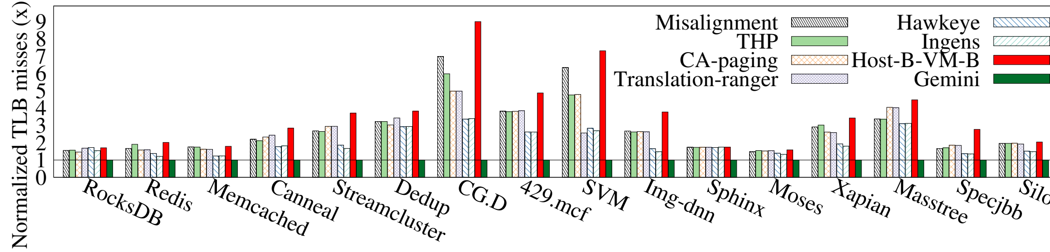

Figure 11. *TLB misses of different systems when workload runs in a clean slate VM.* TLB misses are normalized to those of Gemini.

of forming well-aligned huge pages for virtualized clouds. This can greatly increase tail latency. Second, we find that the tail latency is vulnerable to TLB misses and page walk overhead.

When memory is unfragmented, Gemini, Ingens, and HawkEye perform similarly on some workloads (e.g., SVM and CG.D), as shown in Figure 8, Figure 9, and Figure 10. We find that these workloads usually allocate large memory regions with static arrays and use them uniformly, such that the dense and uniform memory access patterns make these systems also form many well-aligned huge pages. However, for some other workloads (e.g., Redis and RocksDB), they allocate large memory (more than 10GB) gradually and use dynamic data structures to save temporary data, and their memory patterns are more complex and quickly cause memory fragmentations. For these workloads, Gemini outperforms Ingens and HawkEye, no matter memory is fragmented or not.

To further understand the performance of all the evaluated systems, we profile TLB misses when workload is executed in each evaluated system. We show the results in Figure 11. We find that Gemini also incurs extra TLB misses as it does not reserve huge pages and needs a short time to form well-aligned huge pages. Therefore, workload may execute without the well aligned huge pages for a short period. Nonetheless, Gemini saves precious memory space compared to memory reservation approaches. In comparison to Gemini, other systems increase the TLB misses by 2.39x on average. This shows Gemini's effectiveness on reducing TLB misses through efficiently forming well-aligned huge pages. Due to space constraints, we only present TLB misses (Figure 11) and rates of well-aligned huge pages (Table 3) when memory is fragmented.

6.3 Workload Performance in An Reused VM

Figure 12, Figure 13, and Figure 14 show all the evaluated systems' throughputs, mean latencies, and 99th tail latencies, respectively, when workload runs in an reused VM. On average, Gemini outperforms Host-B-VM-B by 1.65x and reduces the mean latency and the tail latency by 32% and 26%, respectively, compared to Host-B-VM-B.

Among all the other systems, Ingens achieves the second largest performance on average. Compared to Ingens,

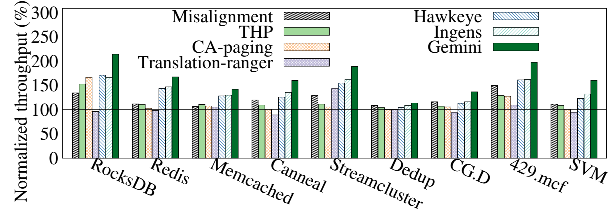

Figure 12. *Throughputs of different systems when workload runs in an reused VM.* Throughputs are normalized to those of Host-B-VM-B.

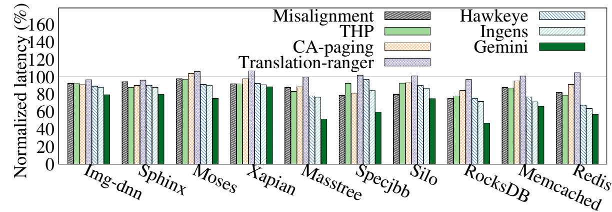

Figure 13. *Mean latencies of different systems when workload runs in an reused VM.* Mean latencies are normalized to those of Host-B-VM-B.

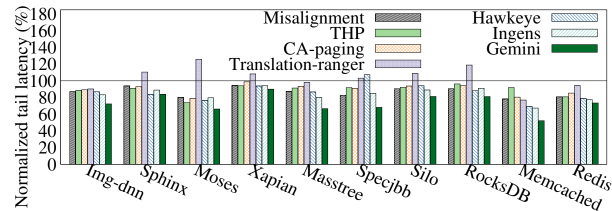

Figure 14. *Tail latencies (99th) of different systems when workload runs in an reused VM.* Tail latencies are normalized to those of Host-B-VM-B.

Gemini offers 25% more throughput, 14% lower mean latency, and 11% lower tail latency. Translation-ranger shows the worst performance among the evaluated systems, due to the costly page migrations overhead. HawkEye and Ingens show comparable performance, and their performance outperforms THP on average. HawkEye and Ingens are optimized based on THP especially for asynchronous and utilization-based promotion, and more fair huge pages promotion. Misalignment shows better performance compared to Host-B-VM-B, as it only forms huge pages in the host OS. CA-paging shows better average performance compared to Host-B-VM-B because it tries to allocate contiguous memory space in both guest- and host-level to predict memory address translation between guest virtual address and

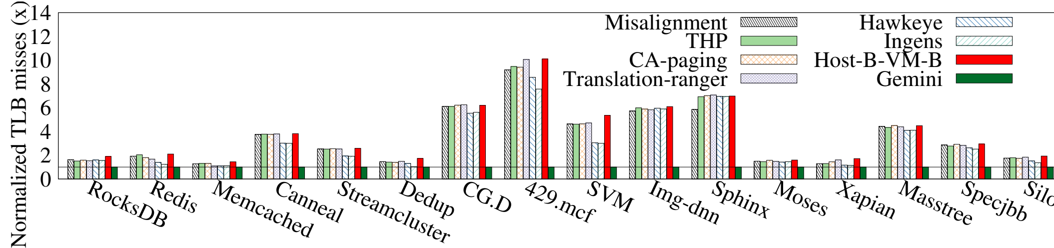

Figure 15. *TLB misses of different systems when workload runs in an reused VM.* TLB misses are normalized to those of Gemini.

workloads	THP	CA-paging	Trans.-ranger	HawkEye	Ingens	Gemini
Img-dnn	52%	58%	35%	57%	51%	84%
Sphinx	64%	57%	36%	61%	63%	89%
Moses	59%	54%	44%	65%	52%	94%
Xapian	55%	59%	31%	52%	58%	96%
Masstree	56%	66%	30%	51%	63%	84%
Specjbb	64%	51%	40%	59%	67%	88%
Silo	59%	61%	36%	63%	60%	75%
RocksDB	64%	59%	31%	67%	51%	83%
Redis	61%	55%	32%	61%	64%	81%
Memcached	60%	56%	37%	63%	66%	86%
Canneal	56%	64%	36%	58%	69%	92%
Streamcluster	52%	62%	38%	55%	68%	90%
dedup	50%	64%	33%	51%	50%	79%
CG.D	58%	68%	37%	68%	60%	98%
429.mcf	51%	58%	34%	52%	62%	93%
SVM	59%	62%	32%	60%	66%	99%

Table 4. Rates of well-aligned huge pages in different systems when workload runs in an reused VM.

host physical address, such that it may form well-aligned huge pages and mitigate address translation overhead.

To pinpoint the performance advantage of Gemini compared to other systems, we profile TLB misses and rates of reusing the well-aligned huge pages that have been formed. We show the profiling results in Figure 15 and Table 4, respectively. Gemini shows low TLB misses for two main reasons. First, Gemini's huge booking (HB) and enhanced memory allocator (EMA) mechanisms greatly increase the rate of well-aligned huge pages. Second, Gemini's huge bucket mechanisms help reuse 88% of the well-aligned huge pages on average, after the completion of SVM workload. Relative to Gemini, other systems increase TLB misses by 4.6x on average. For these systems, after SVM terminates, the allocated huge pages are freed and returned to the guest OS. These huge page sized memory regions may be reallocated for other base page (4KB) allocations (e.g., small memory allocations in Xapian). This may break the well-aligned huge pages created by the SVM workload, causing serious performance degradation.

6.4 Performance Breakdown

Figure 16 shows the performance breakdown of Gemini under memory fragmentation. Gemini's EMA/HB and huge bucket contribute to the whole throughput by 66% and 34% on average, respectively. EMA/HB contribute the most as they

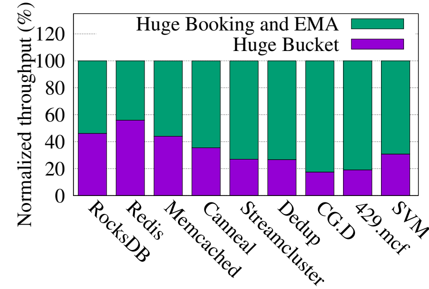

Figure 16. *Performance breakdown of Gemini.* Performance of each part is normalized to the total performance of Gemini.

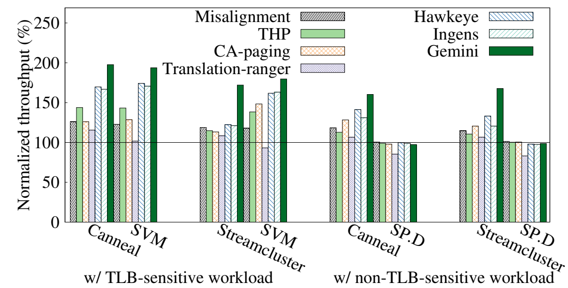

Figure 17. *Throughputs of different systems when different workloads in VMs are collocated in the same server.* Throughputs are normalized to those of Host-B-VM-B.

are the foundation of effectively forming well-aligned huge pages with low overhead. Huge bucket further improves Gemini's efficiency on forming well-aligned huge pages. For some workloads (e.g., CG.D and SVM), EMA/HB outperform huge bucket, as these workloads usually allocate a chunk of memory region and do not frequently free and reuse memory. For some other workloads (e.g., Redis and RocksDB), EMA/HB and huge bucket perform similarly because the well-aligned huge pages are frequently created, freed, and reused during the execution of the workload.

6.5 Applicability and Overhead

To evaluate Gemini's applicability and overhead, we collocate two virtual machines (VMs) on the server. Each VM has 16 virtual CPUs (vCPUs). In each VM, we run either a TLB-sensitive application or a non-TLB-sensitive application; each application has 16 threads. Figure 17 and Figure 18 show throughputs and mean latencies of all the evaluated systems, respectively. On average, Gemini performs the best among

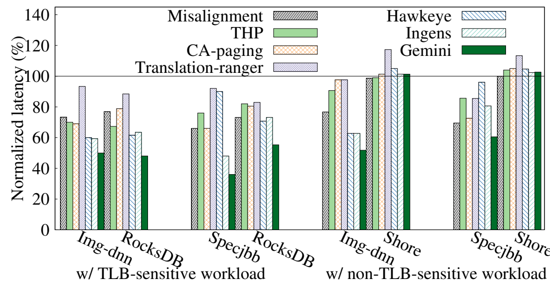

Figure 18. *Mean latencies of different systems when different workloads in VMs are collocated in the same server.* Latencies are normalized to those of `Host-B-VM-B`.

all the systems. Compared to Gemini, other systems offer 37% lower throughput and 19% higher mean latency on average. This shows Gemini's effectiveness on multi-threaded applications, multiple processors, and multiple VMs consolidated on the same server. Gemini allows multiple threads to book well-aligned huge pages concurrently through efficiently batching memory allocations. For multiple processors (multiple NUMA nodes), Gemini searches the Gemini *contiguity list* and finds the contiguity free memory space in the NUMA node that is close to the thread. Besides, Gemini's huge bucket mechanisms also mitigate interferences between small and large memory allocations.

Figure 17 and Figure 18 also show Gemini does not introduce much performance overhead (2% on average). When TLB-sensitive workload is collocated with non-TLB-sensitive workload (i.e., `NPB SP.D` and `Shore`), there is almost no space for Gemini to improve the performance of `NPB SP.D` and `Shore`. For these workloads, Gemini's performance drops by up to 3%, compared to vanilla Linux/KVM. This shows Gemini introduces negligible overhead.

7 Related Work

Huge pages. Existing works [17–19, 26, 27] show that TLB misses and address translation overhead have become the performance bottleneck for modern big memory workloads. Huge pages have been extensively studied to reduce such overhead in native systems [1, 5, 15]. Many systems have actually supported huge pages for a long time (e.g., Linux `Libhugetlbfs` [55, 56]). They mainly require users to manually control the use of huge pages.

To support transparent huge pages, many research works have been proposed [8, 16]. `Ingens` [1] identifies several issues in Linux transparent huge page (i.e., Linux `THP`) mechanisms, such as long latency caused by synchronous page fault and unfair allocation of huge pages among processes. It proposes asynchronous huge page allocation, utilization-based huge page promotion, and a share-based policy to allocate huge pages fairly. `HawkEye` [8] identifies several suboptimal issues in `Ingens` [1], and proposes new mechanisms to address these issues through measuring page translation overhead with hardware performance counters. Temeraire [18] enables aggressive huge pages allocations in TCMalloc [57].

FreeBSD uses a reservation-based memory allocator [5, 16], which allocates a 2MB-aligned physical memory region upon the first page fault and creates the huge page mapping after the entire memory region is accessed. Navarro et al.[5] optimizes the reservation-based huge page management with new mechanisms such as multiple huge page sizes support and contiguity-aware page replacement algorithm to control memory fragmentations. Zhu et al.[16] comprehensively analyze huge pages mechanisms and propose Quicksilver to optimize memory bloat and fragmentation problems.

Optimizing address translation in virtualized systems. Merrifield et al.[12] study workload performance and TLB misses when base pages and huge pages are used in virtualization environments; they mention that both guest- and host-level must map the page at 2MB to allow the processor to use a 2MB TLB entry. `CA-paging` [26] mitigates the address translation overhead through software and hardware codesign. GLUE [13, 14] mitigates the page splintering problem [9, 58–66] in virtualized systems.

Other works. `Translation-ranger` [27] is an OS support to improve memory contiguity and reduce the number of contiguous memory regions to reduce TLB burden. RMM [33] enables ranges of an arbitrary number of virtually and physically contiguous pages to increase TLB reach. To realize the idea, it adds a hardware range TLB and an range page table to enable range address translation.

Gemini's novelty. Existing works mainly target the efficiency of huge page mechanisms in native systems. They are complementary to Gemini. Gemini targets the huge page misalignment problem in virtulized clouds. Previous works [1, 27, 49] show that memory quickly fragments in multi-tenant virtualized cloud environments, and thus this problem can easily happen. Gemini addresses the problem through effectively forming well-aligned huge pages on virtualized clouds.

8 Conclusion and Future Work

This paper identifies the huge page misalignment problem that can significantly increase TLB misses and degrade application performance on virtualized platforms. It designs Gemini as an effective cross-layer page coalescing solution for this problem. By forming well-aligned huge pages with low overhead, Gemini can substantially reduce TLB misses and improve performance. Our evaluation based on diverse real-world applications shows state-of-the-art page coalescing mechanisms still suffer the huge page misalignment problem, and Gemini can achieve substantially better performance than these mechanisms.

As future work, we plan to test and further improve Gemini under more workload and more complex scenarios. For example, memory deduplication (e.g., Linux KSM), memory ballooning, and swapping are used in virtualization environments to deal with memory pressure. They may demote huge pages that are created by Gemini and reduce the

performance of Gemini. In our current design, we only allow misaligned huge pages and infrequently used huge pages to be demoted when system is under memory pressure. We would like to further study how memory deduplication, ballooning, and swapping may interplay with Gemini when such huge pages run out.

9 Acknowledgments

We thank the anonymous reviewers for their constructive comments, and Dr. Gaël Thomas for his helpful suggestions as the shepherd for this paper.

References

[1] 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, 2016.

[2] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman S Ünsal. Range translations for fast virtual memory. *IEEE Micro*, 36(3):118–126, 2016.

[3] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched address translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1023–1036, 2019.

[4] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices*, 47(4):37–48, 2012.

[5] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review*, 36(SI):89–104, 2002.

[6] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–300, 2020.

[7] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. Fast local page-tables for virtualized numa servers with vmitosis. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[8] Ashish Panwar, Sorav Bansal, and K Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–360, 2019.

[9] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. Proactively breaking large pages to improve memory overcommitment performance in vmware esxi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 39–51, 2015.

[10] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John CS Lui. Smartmd: A high performance deduplication engine with mixed pages. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 733–744, 2017.

[11] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. Ptemagnet: Fine-grained physical memory reservation for faster page walks in public clouds. In *The 26th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021*, 2021.

[12] 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*, pages 25–35, 2016.

[13] Binh Pham, Ján Veselỳ, Gabriel H Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 1–12, 2015.

[14] Binh Pham, Jan Vesely, Gabriel H Loh, and Abhishek Bhattacharjee. Using tlb speculation to overcome page splintering in virtual machines. 2015.

[15] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 679–692, 2018.

[16] Weixi Zhu, Alan L Cox, and Scott Rixner. A comprehensive analysis of superpage management mechanisms and policies. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 829–842, 2020.

[17] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 804–818, 2021.

[18] AH Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 257–273, 2021.

[19] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S McKinley, and Paul Turner. Adaptive huge-page subrelease for non-moving memory allocators in warehouse-scale computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, pages 28–38, 2021.

[20] Intel 64 and ia-32 architectures developer's manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[21] Amd64 architecture programmer's manual. https://developer.amd.com/resources/developer-guides-manuals/.

[22] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 121–131, 2019.

[23] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 26–35, 2008.

[24] The Princeton application repository for shared-memory computers (PARSEC). http://parsec.cs.princeton.edu/, 2010.

[25] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[26] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and exploiting contiguity for fast memory virtualization. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 515–528. IEEE, 2020.

[27] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 698–710, 2019.

[28] Linux Transparent Huge Pages. https://lwn.net/Articles/359158/.

[29] James H. Hester and Daniel S. Hirschberg. Self-organizing linear search. *ACM Computing Surveys (CSUR)*, 17(3):295–311, 1985.

[30] Kenneth C Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623–624, 1965.

[31] DE Knuth. The art of computer programming: Fundamental algorithms, volume 1. addisonwesley. *Reading, Mass.,*, 1997.

[32] David G. Korn and Kiem-Phong Bo. In search of a better malloc. In *Proceedings of the Summer 1985 USENIX Conference*, pages 489–506. USENIX, 1985.

[33] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D Hill, Kathryn S McKinley, Mario Nemirovsky, Michael M Swift, and Osman Ünsal. Redundant memory mappings for fast access to large memories. *ACM SIGARCH Computer Architecture News*, 43(3S):66–78, 2015.

[34] Xiaolin Wang, Taowei Luo, Jingyuan Hu, Zhenlin Wang, and Yingwei Luo. Evaluating the impacts of hugepage on virtual machines. *Science China Information Sciences*, 60(1):012103, 2017.

[35] A deep network handwriting classifier. https://github.com/xingdi-eric-yuan/multi-layer-convnet.

[36] Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, and Joe Woelfel. Sphinx-4: A flexible open source framework for speech recognition, 2004.

[37] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics, 2007.

[38] Xapian project. https://github.com/xapian/xapian.

[39] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[40] The SPECjbb benchmark. https://www.spec.org/jbb2015/.

[41] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[42] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35, 2009.

[43] RocksDB NoSQL Storage System. https://rocksdb.org/.

[44] Redis In-memory Key-Value Database. http://redis.io/.

[45] Memcached Key-Value Store. https://memcached.org.

[46] NASA Parallel Benchmarks. http://www.nas.nasa.gov/publications/npb.html.

[47] SPEC CPU 2006. https://www.spec.org/cpu2006/.

[48] Ching-Pei Lee and Chih-Jen Lin. Large-scale linear ranksvm. *Neural computation*, 26(4):781–817, 2014.

[49] Jean Araujo, Rubens Matos, Paulo Maciel, Rivalino Matias, and Ibrahim Beicker. Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure. In *Proceedings of the Middleware 2011 Industry Track Workshop*, pages 1–7, 2011.

[50] Source code of CA-Paging. https://github.com/cslab-ntua/contiguity-isca2020.

[51] Memory is not released after workloads in VMs free memory. https://bugzilla.redhat.com/show_bug.cgi?id=995420.

[52] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. Demand-based coordinated scheduling for smp vms. In *ACM SIGPLAN Notices*, volume 48, pages 369–380. ACM, 2013.

[53] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scaling guest {OS} critical sections with ecs. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 159–172, 2018.

[54] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down tlb shootdowns! In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–14, 2020.

[55] Persistent huge pages in Linux. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[56] Huge pages part 2: Interfaces. https://lwn.net/Articles/375096/.

[57] Tcmalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[58] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. *ACM SIGOPS Operating Systems Review*, 43(3):27–36, 2009.

[59] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '05, 2005.

[60] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems*

Design & Implementation - Volume 2, NSDI'05, 2005.

[61] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, OSDI '02, pages 181–194, USA, 2002. USENIX Association.

[62] VMWARE Large Page Performance Study. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/large_pg_performance.pdf.

[63] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. *ACM SIGARCH Computer Architecture News*, 41(3):237–248, 2013.

[64] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on {NUMA} systems. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 231–242, 2014.

[65] Yuan Xie. Modeling, architecture, and applications for emerging memory technologies. *IEEE design & test of computers*, 28(1):44–51, 2011.

[66] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. Hub: Hugepage ballooning in kernel-based virtual machines. In *Proceedings of the International Symposium on Memory Systems*, pages 31–37, 2018.