Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters

Kaiyang Zhao
Carnegie Mellon University
kaiyang2@cs.cmu.edu

Kaiwen Xue
Carnegie Mellon University
kaiwenx@andrew.cmu.edu

Ziqi Wang
Carnegie Mellon University
ziqiw@andrew.cmu.edu

Dan Schatzberg
Meta Platforms, Inc.
dschatzberg@meta.com

Leon Yang
Meta Platforms, Inc.
lnyng@meta.com

Antonis Manousis
Meta Platforms, Inc.
amanousis@meta.com

Johannes Weiner
Meta Platforms, Inc.
jweiner@meta.com

Rik van Riel
Meta Platforms, Inc.
riel@meta.com

Bikash Sharma
Meta Platforms, Inc.
bsharma@meta.com

Chunqiang Tang
Meta Platforms, Inc.
tang@meta.com

Dimitrios Skarlatos
Carnegie Mellon University
dskarlat@cs.cmu.edu

ABSTRACT

The unabating growth of the memory needs of emerging datacenter applications has exacerbated the scalability bottleneck of virtual memory. However, reducing the excessive overhead of address translation will remain onerous until the physical memory contiguity predicament gets resolved. To address this problem, this paper presents Contiguitas, a novel redesign of memory management in the operating system and hardware that provides ample physical memory contiguity. We identify that the primary cause of memory fragmentation in Meta's datacenters is unmovable allocations scattered across the address space that impede large contiguity from being formed. To provide ample physical memory contiguity by design, Contiguitas first separates regular movable allocations from unmovable ones by placing them into two different continuous regions in physical memory and dynamically adjusts the boundary of the two regions based on memory demand. Drastically reducing unmovable allocations is challenging because the majority of unmovable pages cannot be moved with software alone given that access to the page cannot be blocked for a migration to take place. Furthermore, page migration is expensive as it requires a long downtime to (a) perform TLB shootdowns that scale poorly with the number of victim TLBs, and (b) copy the page. To this end, Contiguitas eliminates the primary source of unmovable allocations by introducing hardware extensions in the last-level cache to enable the transparent and efficient migration of unmovable pages even while the pages remain in use.

We build the operating system component of Contiguitas into the Linux kernel and run our experiments in a production environment at Meta's datacenters. Our results show that Contiguitas's OS component successfully confines unmovable allocations, drastically reducing unmovable 2MB blocks from an average of 31% scattered across the address space down to 7% confined in the unmovable region, leading to significant performance gains. Specifically, we show that for three major production services, Contiguitas achieves end-to-end performance improvements of 2-9% for partially fragmented servers, and 7-18% for highly fragmented servers, which account for nearly a quarter of Meta's fleet. We further use full-system simulations to demonstrate the effectiveness of the hardware extensions of Contiguitas. Our evaluation shows that Contiguitas-HW enables the efficient migration of unmovable allocations, scales well with the number of victim TLBs, and does not affect application performance. We are currently in the process of upstreaming Contiguitas into Linux.

CCS CONCEPTS

• **Software and its engineering → Operating systems**; **Virtual memory**; • **Computer systems organization → Architectures**.

KEYWORDS

Datacenters; Operating Systems; Memory Management; Virtual Memory

ISCA '23, June 17–21, 2023, Orlando, FL, USA.

ACM ISBN 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589079

ACM Reference Format:
Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik van Riel, Bikash Sharma, Chunqiang Tang, and Dimitrios Skarlatos. 2023. Contiguitas: The Pursuit of Physical Memory Contiguity in Datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589079

1 INTRODUCTION

Memory capacity has increased dramatically over the last decades, yet modern operating systems have throughout stuck with a small base page size. This divergence has resulted in excessive management overhead for memory-intensive applications. In particular, virtual memory implementations are plagued by expansive page table trees, and a corresponding appetite for hardware TLB capacity that is difficult to satiate.

Even with architectural innovations such as larger and multi-level TLBs and page walk caches [4, 14–16, 21–24, 29, 60, 82], applications today suffer a substantial performance penalty due to TLB misses. Google's internal profiling revealed that approximately 20% of cycles are stalled on TLB misses [48]. Unfortunately, this problem is only bound to get worse due to: i) the inherent hardware limits of TLB scaling, already surpassing L2 cache latencies [112], ii) terabyte-scale memory capacity through technologies like CXL [56, 93], iii) additional levels of page tables [51], iv) the increase of memory-intensive applications, and v) upcoming confidential computing platforms that place security checks at page granularity during address translation [10, 53].

A large body of prior research has focused on reducing the address translation overhead [1–3, 14, 16, 20, 23, 30, 36, 45, 48, 60, 62, 71, 77, 83, 92, 95, 96, 98–100, 108, 115]. Conceptually, we can separate prior work based on the amount of physical memory contiguity required and how it is exploited. On the one hand, earlier works propose leveraging physical memory contiguity to back the application dataset [4, 16, 36, 37, 60, 61, 82]. These approaches create range-based translations, similar to segments, that map large contiguous regions of virtual memory to equally large contiguous physical memory. Their goal is to ultimately reduce the number of TLB entries needed. However, they face the fundamental challenge that it is very hard to create multi-gigabyte contiguous physical address ranges to cover the complete application dataset.

On the other hand, another line of research has explored alternative page table structures such as hashed page tables [32, 47, 58, 59, 95, 99, 101, 115]. Indeed, recent work [95, 98, 99, 115] has solved some of their traditional shortcomings [15, 33, 41, 49, 50]. Such solutions aim to replace sequential multi-level page tables and drastically reduce the cost of page walks by accelerating page table accesses. Notably, they relax the physical memory contiguity requirements to apply not on the whole dataset, but only on the page table organization. However, they impose strict requirements for physical memory contiguity availability on the critical path of page table creation. As a result, such approaches still remain challenging to adopt. Several other architectural extensions that implicitly rely on contiguity [44, 72, 82, 108, 114] are hindered by the same fundamental challenge.

Today's operating systems, such as Linux, have mostly relied on 2 MB huge pages to land performance improvements. The primary mechanism to leverage huge pages is Transparent Huge Pages (THP) [105] that *opportunistically* try to provide 2 MB pages. Unfortunately, in today's systems, finding physical contiguity even for 2 MB pages is often hard due to memory fragmentation [38, 40, 43, 64, 77, 78, 83]. THPs have also been under scrutiny due to their performance implications such as latency spikes and memory bloating [9, 25, 26, 38, 43, 66, 76]. Alternative approaches, such as userspace allocators [48, 67, 68] still rely on the OS to provide physical contiguity and larger mappings.

Addressing excessive memory management overhead will require a fundamental shift towards larger page granularities. However, fragmentation remains as the primary obstacle. Illuminator [78] is a major prior work that tries to address fragmentation. Within a 2 MB block, it prevents mixing *unmovable* allocations and *movable* ones that the kernel can move on-demand. This work, while innovative, has some key limitations.

First, the fundamental problem of unmovable allocations, i.e., that the OS cannot move them after their allocation, remains. Second, avoiding mixing movable and unmovable allocations within 2 MB blocks still fragments the address space, prevents large contiguous regions from being formed, and consequently blocks larger translations. This is because it limits the maximum available contiguity to at most 2 MB. For example, a single unmovable 4 KB page can render a 1 GB region unmovable; as a result just 0.19% of 2MB unmovable allocations can fragment the whole of memory irrecoverably. Third, the effectiveness of defragmentation is severely hindered by the presence of unmovable allocations [114]. Overall, despite significant efforts in virtual memory research, eliminating the address translation overhead will remain onerous in real world environments until the physical memory contiguity predicament gets resolved.

1.1 This Paper: Ample Physical Memory Contiguity by Design.

In this work, we start with a detailed investigation of physical memory contiguity at hyperscale across Meta's datacenters. We sample servers across the fleet and show that *23% of servers do not even have physical memory contiguity for a single* 2 MB *huge page*. We also find that it is practically impossible to dynamically allocate 1 GB pages in a production environment. Pertinently, fragmentation affects all servers as there is little to no correlation between memory contiguity availability and server up-time. In practice, servers can quickly get heavily fragmented within the first hour after boot-up while the mean server uptime is multiple days or weeks—turning memory fragmentation into a major challenge. Finally, our study exposes unmovable memory allocations as the root cause for the lack of physical memory contiguity. In particular, we identify several sources of unmovable allocations, including networking buffers, slab, filesystem, and page tables.

To address these issues, we introduce Contiguitas with the goal of eliminating fragmentation due to unmovable allocations. Contiguitas separates movable allocations from unmovable ones by placing them into two different continuous regions and dynamically adjusts the boundary of the two regions based on memory demand. To avoid wasting memory in the unmovable region, Contiguitas solves two problems: i) how to dynamically resize the unmovable region and place unmovable allocations; and ii) how to drastically reduce unmovable allocations. For the first problem, Contiguitas performs resizing by tracking the demand for unmovable allocations. Moreover, it reduces internal fragmentation of the unmovable region by differentiating different types of unmovable allocations.

For the second problem, Contiguitas focuses on unmovable allocations that cannot be moved with software alone because access

to the page cannot be blocked for a migration to take place. At Meta, networking allocations account for 73% of unmovable pages (Section 2.5). We expect unmovable allocation to become an increasingly bigger problem. This is because of new I/O technologies such as kernel-bypass and RDMA for networking and storage, GPUs, and other accelerators that heavily really on unmovable pages.

To this end, Contiguitas introduces a set of surgical hardware extensions in the last-level cache (LLC) that enable the *transparent* migration of unmovable pages while in use. Contiguitas's design builds off of two ideas: First, Contiguitas introduces migration mappings in the LLC, enabling hardware to redirect traffic to the appropriate cache line of each page based on the progress of the migration. Second, Contiguitas relaxes the TLB shootdown operation from being synchronous and requiring acknowledgements from all victim TLBs to a local TLB invalidation that can be performed by each core independently and in a lazy manner. Naturally, movable page migrations can also benefit from this hardware support.

We build the OS component of Contiguitas into the Linux kernel and run our experiments in Meta's production environment. Our results show that this component successfully confines unmovable allocations, drastically reducing unmovable 2MB blocks from an average of 31% scattered across the address space down to 7% confined in the unmovable region, leading to significant performance gains. Specifically, we show that for three major production services, Contiguitas achieves performance improvements between 2-9% for partially fragmented servers that represent the majority of the servers, and between 7-18% for highly fragmented servers representative of nearly a quarter of the fleet at Meta. Notably, Contiguitas's contiguity gains enable Web, one of Meta's largest services, to dynamically allocate 1 GB huge pages, leading to a 7.5% performance win that is unattainable with 2 MB pages alone. We use full-system simulations to demonstrate the effectiveness of the hardware extensions of Contiguitas. Our evaluation shows that Contiguitas-HW enables the efficient migration of unmovable allocations while scaling the number of victim TLBs and does not affect application performance. We are currently in the process of upstreaming Contiguitas into Linux.

2 MEMORY CONTIGUITY CHALLENGES AND OPPORTUNITIES IN DATACENTERS

In this section, we first provide a brief overview of memory management of modern operating systems. Then we showcase the challenges and opportunities of memory contiguity across Meta's datacenters through a detailed study of i) memory capacity and TLB trends, ii) performance implications of address translation, iii) memory fragmentation, and iv) unmovable allocations and their sources.

2.1 Memory Management

Multiple Page Sizes. Modern operating systems, such as Linux, manage memory in small 4 KB page granularity. The primary reason behind this decision is to reduce memory bloating and expensive IO operations during paging. However, 4 KB pages cause significant address translation overhead due to the limited capacity of the TLBs and expensive page walks in the event of a TLB miss. To that end, huge pages of 2 MB and 1 GB have been retrofitted in the kernel with additional hardware support. Huge pages reduce the number of translation entries and further shorten page walks.

Linux has two primary mechanisms to allow applications to leverage huge pages. The HugeTLB subsystem allows a system administrator to allocate a certain number of persistent huge pages which can then be explicitly mapped by userspace applications. HugeTLB requires careful coordination between system administrators and application developers to ensure a proper number of huge pages are available.

Alternatively, Transparent Huge Pages (THP) [105] can be used to allocate huge pages transparently to the application. THPs only support 2MB pages and opportunistically assign huge pages to applications, either in page fault handling or through background page promotion. The OS forms huge pages on top of contiguous physical memory regions that it keeps on free lists. However, userspace memory pages, kernel structures, and other sources can cause a fragmented physical address space with no huge pages available.

Memory Fragmentation. Memory fragmentation can be alleviated by memory compaction, also known as defragmentation. Memory compaction [28] tries to consolidate physical pages by moving them around, freeing up contiguous memory areas for huge page allocations. During movement, the page becomes temporarily unavailable until the page is copied to the new location and the new mapping is established.

Page Migration and TLB shootdowns. Figure 1 shows the process of page migration. A translation for a page is cached in the TLBs. However, in contemporary processors TLBs are not cache-coherent. Hence, the OS needs to invalidate all potential *victim* TLBs of cores that might be caching the translation. First, in Step ① the OS running on top of the *initiator* core clears the present bit of the page table entry of the page under migration. Then, the initiator invalidates the entry from the local TLB and initiates the TLB shootdown procedure, in Step ②. Specifically, the OS sends interprocessor interrupts (IPIs) in Step ③ to all remote victim cores. Each core receiving an IPI invokes an interrupt handler that flushes their private TLB, shown in Step ④, and then sends an acknowledgment in Step ⑤. After the initiator receives all the required acknowledgments, it performs the copy of the page in Step ⑥. Finally, it updates the PTE in Step ⑦.

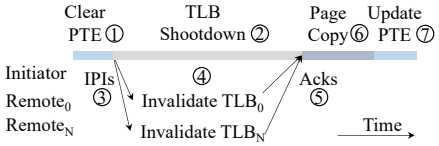

Figure 1: Page migration procedure.

TLB shootdowns are very costly as they scale poorly with the number of involved cores and may require several thousands of cycles to complete [7, 8, 12, 17, 63, 63, 90, 109]. Upcoming hardware [6, 11, 54] aims to reduce the TLB invalidation overhead, but a page still becomes unavailable during migration. The same steps are followed by processors that make use of IOMMUs with IOTLBs [5, 55, 80] and NICs with private TLBs [84, 104].

Unmovable Allocations. An impediment to memory compaction is *unmovable allocations* that cannot be migrated. Unmovable allocations exist across different operating systems, called wired

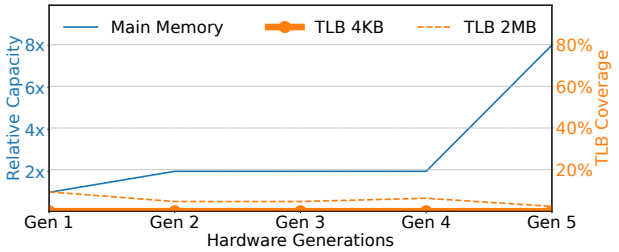

Figure 2: Memory and TLB coverage of computing hardware across generations.

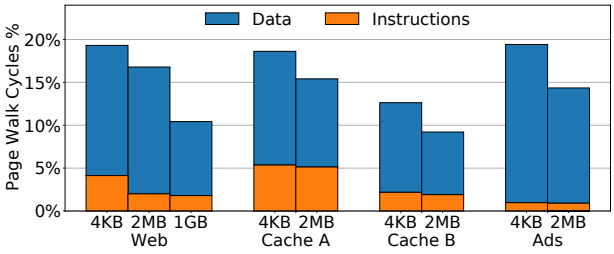

Figure 3: Percentage of cycles lost due to page walks.

allocations in FreeBSD, and non-paged in Windows [74, 91, 114]. While there are many sources of unmovable allocations such as slab, page tables, and other kernel structures, they can be categorized into two types. The first type of unmovable allocations are formed because the kernel opts for faster translation of kernel objects through a simple offset into a linear map [107]. Addresses to these data structures may be stored all over memory. As a result, such data structures cannot be moved.

Other types of unmovable allocations are related to I/O and the IOMMU, including memory for kernel-bypass networking and storage, GPUs, accelerators, and in general operations that mark pages as busy and thus unmovable. Such allocations are impossible to move with software alone. The reason is that the software cannot atomically perform both the translation update and the page copy. Hence, the only way for software to migrate a page without potential spurious writes to the page taking place during migration is to first block access to the page, perform the copy, and then update the translation. However, access to such unmovable pages cannot be blocked as incoming operations from a device need to be serviced. Even if access to the page could be blocked, page migration itself is expensive because it requires a long downtime to (a) perform a TLB shootdown procedure that scales poorly with the number of victim TLBs, and (b) copy the page.

2.2 Memory Capacity and TLB Trends

Trends in Meta's hardware show increases in memory capacity without comparable increases in TLB capacity. This discrepancy puts increased emphasis on contiguity availability to reduce address translation overheads.

Figure 2 shows the relative increase of memory capacity and TLB coverage of computing infrastructure across hardware generations in Meta's datacenters. The x-axis shows different hardware generations. The *Gen-1* hardware is near its end of life while *Gen-4 and Gen-5* are expected to be deployed in the near future. We normalize the results based on the first generation. Memory capacity is bound to increase by almost 8×. However, the number of TLB entries and consequently TLB coverage remain stagnant. Specifically, the number of TLB entries has steadily remained in the range of a few thousands in the last few generations. As a result, TLB coverage with contemporary 4 KB pages, and even larger 2 MB pages will be significantly insufficient. 1 GB pages do provide sufficient coverage that is larger than the main memory capacity of *Gen-5* hardware. Reducing translation overheads will require ever larger page sizes, and hence contiguity, in hand with techniques to reduce page walk latency.

2.3 Lost Cycles Due to Lack of Contiguity

To quantify the potential of memory contiguity we select a few representative services across Meta's fleet and measure the impact of contiguity. We perform measurements using: i) only 4 KB pages, ii) 2 MB pages, iii) 2 MB and 1 GB. For 2 MB we use THP and further reserve huge pages through HugeTLB [106] for services that have already been optimized to support HugeTLB. To leverage 1 GB pages we solely rely on HugeTLB. We focus the evaluation on the most widely used machine type which has 64GB memory. At Meta, services are deployed directly on Linux servers within containers. Furthermore, services are optimized heavily to fit within the available memory. Hence, the results provide a lower bound of the potential address translation overhead and gains from larger page sizes—we expect such overheads to be further exacerbated as memory capacity increases.

Figure 3 shows page walk cycles as percentages of the total cycles, due to *Data* and *Instructions*, using performance counters of production workloads. For *Data* we show the aggregate of loads and stores. We observe that page walk cycles can account for close to 20% of total cycles in Meta's datacenters. Memory contiguity has the potential to significantly alleviate TLB overheads of both instructions and data. Focusing on Web, one of the largest services within Meta, we see that 2 MB huge pages can halve the number of instruction page walk cycles. Notably, while 2 MB pages offer little improvement for data page walk cycles, 1 GB huge pages have a major impact by reducing such cycles from 14% down to 8%.

2.4 Memory Fragmentation

Memory fragmentation prevents the OS from creating large contiguous regions. To quantify fragmentation across Meta's fleet, we randomly sample tens of thousands of 64GB servers in production independently of workloads and perform a full scan of each server's physical memory. Figure 4 shows the cumulative distribution function (CDF) of memory contiguity as a percentage of free memory at the 2 MB, 4 MB, 32 MB and 1 GB allocation levels. Servers having less than 1 GB of free memory are filtered out. We observe that memory fragmentation is severe across the fleet i.e., 23% of the servers do not have enough contiguity for even a single 2 MB allocation. This number increases to 59% for 32 MB allocations. Dynamically allocating 1 GB pages is practically impossible in production.

Correlation With Uptime. It is commonly assumed that memory fragmentation is correlated with server up-time [78, 114]. In particular, freshly brought-up servers are expected to have an abundance

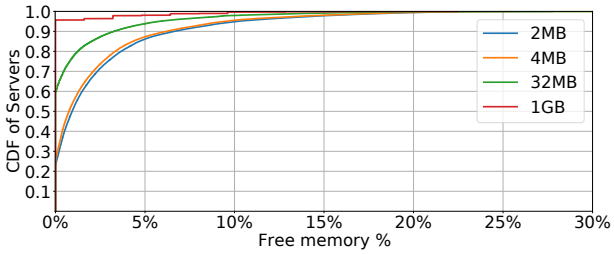

Figure 4: Contiguity availability as the percentage of free memory.

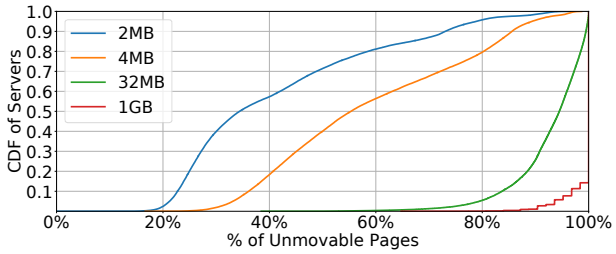

Figure 5: Distribution of unmovable pages in contiguous regions of 2MB, 4MB, 32MB, and 1GB.

of available memory contiguity that *slowly* deteriorates as workloads execute. Interestingly, our analysis showed there is little to no correlation between memory contiguity availability and server uptime, with the Pearson correlation coefficient between server uptime and the number of free 2 MB pages being only 0.00286. We further looked into servers with only a few hours of uptime and discovered that the correlation still remained weak, with a coefficient of 0.16. Finally, we performed a set of experiments that indicated that servers can get highly fragmented within the first hour of running workloads. Given that the mean uptime of Meta's servers is multiple days or weeks, memory fragmentation affects nearly all servers.

2.5 Unmovable Memory Allocations

Despite high amounts of fragmentation, the kernel could succeed in allocating huge pages by compacting allocated pages into fewer contiguous regions. Unmovable allocations inherently limit memory contiguity as they impede memory compaction. To quantify unmovable allocations we follow the same process for studying memory fragmentation. Figure 5 shows the percentage of 2 MB unmovable pages relative to the total memory. As we can see, a median server has 34% of its memory occupied by unmovable allocations. As a result, a significant portion of the memory cannot be used for huge pages, and forming any larger contiguous regions (at 4 MB, 32 MB or 1GB) will be even harder. Furthermore, the variance across machines is a major obstacle for using huge pages. At Meta it is desirable to treat the servers interchangeably so that a workload can potentially land on any available server in the fleet. In practice, for some critical services that depend on the existence of sufficient huge pages, automatic server reboots are used to resolve high fragmentation.

Furthermore, we identified that unmovable allocations are *scattered* across the address space. The median ratio of the number of unmovable 4 KB pages over the total number of pages is only 7.6%, but it makes 34% of 2MB pages unmovable, showing that scattering is greatly exacerbating the unmovable memory issue.

To identify the sources of unmovable allocations, we track all allocations marked as unmovable and backtrace their allocation sources. Figure 6 shows the breakdown of various sources of unmovable allocations. Networking-related operations are a major source of unmovable allocations, accounting for more than 73%. Such allocations include send and receive buffers maintained by the OS. These buffers carry the data received or to be sent through the processing of different layers of the networking stack, from the applications that own the sockets down to NICs [46]. In Meta's environment the number of networking-related pages constantly remains high. More importantly, we expect unmovable networking allocations to become an increasingly bigger problem for two reasons. First, with the adoption of high bandwidth NICs [103] and the increased number of queues based on the number of cores, the number of networking allocations are expected to increase drastically. Second, with kernel bypass and RDMA technologies [34, 42, 73, 85, 117] in datacenters, networking pages are pinned to memory and remain unmovable for the lifetime of an application.

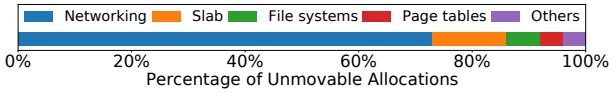

Figure 6: Sources of unmovable allocations.

The second source, which accounts for 12%, is related to slab allocations. The slab allocator is the small object allocator in the Linux kernel that packs objects in pages obtained from the page allocator. It is the primary memory source for kernel data structures which cannot be moved. File systems frequently allocate pages as buffers for compression and decompression. Page tables are used to store translation entries from virtual addresses to physical addresses. About 4% of allocations are related to other sources.

In the future, we expect additional sources of unmovable memory driven by the increased deployment of heterogeneous hardware such as GPUs, accelerators, and other devices that rely on unmovable allocations. As a result, managing unmovable allocations is critical for efficient memory management.

3 CONTIGUITAS DESIGN

The goal of Contiguitas is to provide ample physical memory contiguity by reducing memory fragmentation due to unmovable allocations. To that end, Contiguitas redesigns memory management in the OS to confine unmovable allocations and completely separate them from movable ones. In addition, Contiguitas drastically reduces unmovable pages in datacenters. Specifically, Contiguitas introduces a set of hardware extensions in the last-level cache (LLC) that enable the transparent migration of unmovable pages while in use.

3.1 Overview

Figure 7 provides a high level overview of Contiguitas and how it transforms the physical address space. There are two key design principles guiding Contiguitas. The first one, is to strictly separate unmovable from movable allocations using two dedicated regions

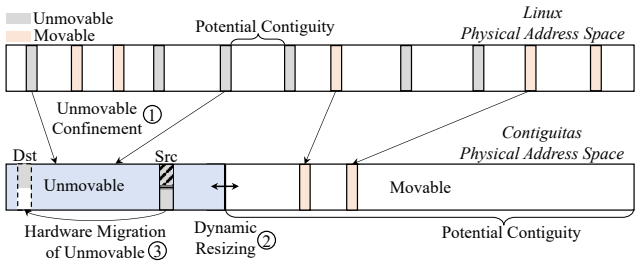

Figure 7: Contiguitas design overview.

and, thus, prevent unmovable allocations from scattering across the address space. The result is shown in Step ①. Next, Contiguitas introduces a dynamic resizing mechanism in Step ②, that monitors the demand for allocations in each region and grows or shrinks the corresponding region accordingly. Contiguitas further reduces internal fragmentation of the unmovable region by placing certain types of allocations away from the region's border.

The second design principle of Contiguitas, is to drastically reduce the amount of unmovable allocations by turning them into movable ones. Our study at Meta's datacenters in Section 2.5 revealed that a significant portion of unmovable allocations are impossible to move as access to the page cannot be blocked for a software migration to take place. Hence, a software solution is inadequate. To this end, Contiguitas introduces a set of hardware extensions in step ③ that enable the transparent migration of unmovable allocations. Naturally, these hardware extensions can further be leveraged by movable pages.

3.2 Confining Unmovable Allocations

Contiguitas separates the physical address space into two continuous regions, one for movable and one for unmovable memory. Allocations are confined in their respective region. Contiguitas categorizes the physical pages based on their addresses and keeps them on distinct free lists for each region. Memory in each region can only be allocated from pages in the free lists belonging to that region. When a page is freed, it is returned back to its respective list. This approach simplifies the critical path of allocations as the OS can quickly pick a free page while avoiding mixing different types of allocations.

Confinement is sufficient for allocations that are deemed unmovable during their whole lifetime. However, some allocations might first be allocated as movable but later become unmovable. For example, due to operations such as memory pinning for zero-copy network send operations. In such scenarios, Contiguitas first migrates them to the unmovable region and then marks them as unmovable. This approach avoids the dynamic pollution of the movable region and subsequent compaction failures.

The major challenge in designing confinement is the sizing of the unmovable region. If it is too big, unused memory in the unmovable region is wasted while there is limited movable memory for the applications, causing frequent reclaims, swapping, or even allocation failures. On the other hand, if the unmovable region is too small, it may fail unmovable allocations. Therefore, Contiguitas needs to dynamically balance the sizes of the movable and unmovable regions.

Dynamic Region Resizing. As the need for movable or unmovable memory may change across applications and their phases, static sizing would be ineffective. To this end, Contiguitas dynamically resizes the two regions to keep the unmovable region small while not negatively affecting application performance. Contiguitas sets up the physical memory layout during system boot-up, and gives the unmovable region a configurable initial size based on the memory capacity of the system and the behavior of workloads in our fleet. For our 64GB servers, we configure this initial size to 4GB. The boundary of the two regions is then tracked in the OS to support resizing. To expand the unmovable region, Contiguitas adjusts the boundary, moves away allocated pages, and takes over space from the movable region, adding pages to the unmovable free list.

Dynamic resizing of the unmovable region introduces three major design challenges. The first challenge is to move resizing operations off the critical path of memory allocation. The second challenge is to choose the placement policy of unmovable allocations within the unmovable region and avoid internal fragmentation. Allocations exhibit various lifetimes and as a result, placing an unmovable allocation with a long lifetime close to the boundary, might unnecessarily block the shrinking of the region even if free space is abundant within the region. The last challenge is to choose the proper size of the unmovable region, which is workload dependent.

Contiguitas performs resizing off the critical path of memory allocation to avoid latency overheads. This is accomplished by monitoring the amount of free memory when periodic memory reclaim is triggered by the kernel. Contiguitas extends reclaim to wake up a kernel thread to perform resizing when the free memory in either region falls below a low-watermark threshold.

In addition, Contiguitas introduces a bias to prefer physical pages further away from the region border. Some unmovable allocations that are inherently long lived, e.g., kernel code pages, are safely placed by Contiguitas early on, at the end of the unmovable region that is farthest from the movable region. On the other hand, pages that are initially in the movable region and later on migrated to the unmovable region often exhibit short lifetimes. In general, Contiguitas prefers allocating pages away from the region border as long as sufficient free space is available. This approach increases the chance that shrinking will be successful. Later on, in Section 3.3 we will further introduce the hardware extensions of Contiguitas that enables the migration of unmovable pages.

Finally, to aide in deciding when and how much to resize, Contiguitas introduces the concept of per-region memory pressure, which builds on top of the kernel's pressure stall information (PSI) used for reclaim [110]. PSI tracks the percentage of time wasted due to lack of free memory (e.g. due to page faults of recently resident memory, or direct reclaim). We extend PSI to track time wasted due to lack of free memory in both the movable and unmovable region separately.

The resizing algorithm is depicted in Algorithm 1. Given per-region pressure, $Pressure_{unmov|mov}$, configurable thresholds, $Threshold_{unmov|mov}$, and coefficients that fine-tune the expansion and shrinkage of the region, c_{ue}, c_{me}, c_{us}, and c_{ms}, the algorithm resizes the unmovable region. Depending on pressure the kernel

Algorithm 1: Region resizing algorithm.

function *Resize*($Pressure_{unmov}$, $Threshold_{unmov}$, $Pressure_{mov}$, $Threshold_{mov}$, c_{ue}, c_{us}, c_{me}, c_{ms})
  if $Pressure_{unmov} \geq Threshold_{unmov}$ *and* $Pressure_{mov} < Threshold_{mov}$ **then**
    // Expand unmovable upon high pressure
    $F \leftarrow \frac{Pressure_{unmov}}{Threshold_{unmov}} \cdot c_{ue} + \frac{Threshold_{mov}}{\max(Pressure_{mov},1)} \cdot c_{me}$
    $U \leftarrow (1+F) \cdot Mem_{unmov}$
  else
    // Shrink for all other cases
    $F \leftarrow \frac{Pressure_{mov}}{Threshold_{mov}} \cdot c_{ms} + \frac{Threshold_{unmov}}{\max(Pressure_{unmov},1)} \cdot c_{us}$
    $U \leftarrow (1-F) \cdot Mem_{unmov}$
  return U;

expands or shrinks the unmovable region. Contiguitas sets parameters for dynamically resizing empirically by observing the patterns for movable and unmovable allocations of the workloads, and tunes the parameters iteratively. Contiguitas uses global parameters that work well for a diverse set of workloads, and we leave automated parameter space search as future work.

3.3 Architectural Support for Transparent Page Mobility

The goal of the hardware extensions of Contiguitas, Contiguitas-HW, is to enable the migration of the subset of unmovable pages that must remain accessible even during migration. Such pages represent a major chunk of allocations in datacenters that is bound to increase in the near future (Section 2.5). Hardware support is required because it is impossible for software to move such pages as it cannot atomically perform both the translation update and the page copy operation. Hence, software has to block access to the page for the duration of page migration in order to avoid spurious writes to it. Even if access to the page could be blocked, software page migration induces a long downtime due to (a) TLB shootdowns that scale poorly with the number of victim TLBs, and (b) the page copy.

To this end, Contiguitas-HW enables transparent page migration while the page remains in use. Such migrations can substantially reduce the size of the unmovable region and lead to more efficient defragmentation and memory management as the vast majority of pages can be moved on demand. While Contiguitas-HW is motivated by unmovable allocations, its design is suitable for both movable and unmovable allocations.

Platform Overview. Figure 8(a) shows a high-level overview of the hardware platform of Contiguitas-HW. The hardware extensions of Contiguitas-HW are located in the LLC. Contiguitas-HW targets environments that are representative of current and upcoming hardware platforms such as those based on CXL [27]. Specifically, Contiguitas-HW targets a multi-core processor with a cache-coherent interconnect. The hardware platform further includes an IOMMU [5, 55] with local TLBs. The IOMMU performs similar duties to the MMU on the CPU side, such as page walks and caching of translations in the TLB for devices. The NIC includes a TLB and is cache-coherent with the LLC. The core performs IOTLB shootdowns by putting invalidation requests onto a queue in memory [55]. Device TLBs cache translations from the IOMMU, and the core can synchronize them with the IOMMU by submitting invalidation requests through the aforementioned queue.

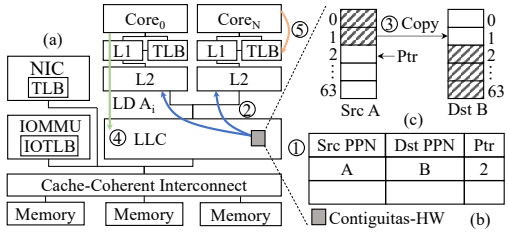

Figure 8: Contiguitas hardware overview. (a) shows the hardware extension of Contiguitas-HW in the LLC. (b) shows the metadata table of Contiguitas-HW. (c) shows the page migration process followed by Contiguitas-HW.

Hardware Operations. At a high level, Contiguitas-HW aliases a physical page under migration with a destination page and redirects appropriate traffic to the destination page based on the progress of the migration.

Specifically, a page migration is initiated in Step ① by the OS that provides the source and destination physical page numbers (PPNs) to the Contiguitas-HW. Contiguitas-HW stores them in a metadata table shown in Figure 8(b). In addition, Contiguitas-HW stores a *Ptr* field that points to the next line to be copied [18, 86, 95]. It effectively tracks the number of cache lines that have been copied.

Before a cache line can be copied, Contiguitas-HW issues *BusRdX* events for both the source and the destination cache lines, shown in Step ②. This step guarantees that the most recent version of the source cache line is located in the LLC and that both source and destination cache lines are invalidated in higher level caches. Next, Contiguitas-HW copies the line from the source to the destination, as shown in Step ③. Then, it increments *Ptr* by one. This process continues until the page is completely copied.

During the migration process, a request, e.g., LDA_i may arrive at the LLC, as shown in Step ④. If the request is for the source page then Contiguitas-HW uses the page offset of the address and compares it to the value of *Ptr*. For addresses less than *Ptr*, the LLC returns the line from the destination page. This is because the cache line has already been migrated. Otherwise, the LLC returns the line from the source page.

When the copy of the page is complete, Contiguitas-HW sets a flag that is periodically checked by the OS. In our design, we opt to perform this check whenever the kernel naturally runs next on each core e.g., due to context switches or system calls. When the OS detects that the flag is set, it updates the page table entry to point to the destination page. Then, the OS performs a local TLB invalidation, shown in Step ⑤. When all TLBs are eventually invalidated, the OS clears the entry from the metadata table in Figure 8(b).

Critically, Contiguitas-HW opts to maintain both mappings concurrently active for the duration of page migration and rely on the LLC to redirect traffic. This approach allows TLBs to switch from the source to the destination mapping without a long downtime, greatly simplifying the TLB shootdown procedure. In Contiguitas-HW, TLB shootdowns do not require inter-processor interrupts (IPIs), as each TLB can be locally invalidated without coordination and synchronous acknowledgements. Similarly, any available core can issue a TLB invalidation to the NIC and the IOMMU. During this process the page under migration is always accessible.

Ultimately, Contiguitas-HW enables the a page under migration to be accessed by the source and destination mappings. Depending on the underlying cache microarchitecture, Contiguitas-HW is amenable to multiple optimizations. Next we discuss two design points that shed light on the trade-offs between noncacheable and cacheable accesses for the page under migration based on underlining microarchitecture mechanisms.

Noncacheable Accesses. One approach to realize the Contiguitas-HW is to rely on noncheable accesses during migration. When a cache line in the source page has been copied, the LLC treats it from then on as noncacheable for the L1 and L2 caches. This optimization is useful because at any point a core may issue requests based on the source mapping found in the TLB while Contiguitas-HW may respond based on the data in the destination mapping.

While it is possible to allow the caching of the line in the private caches, this approach would introduce two implications: First, it would require Contiguitas-HW to keep track of such lines to maintain coherence. Second, writes performed to the source page would have to be propagated to the destination as well. Noncacheables accesses resolve this issue as all traffic is redirected to the LLC.

Noncacheable accesses are already supported by contemporary processors [52]. In existing processors, page table entries maintain a page-level cache disable (PCD) bit that is also installed in the TLB. Requests for a page with the PCD bit set in the TLB are performed as noncacheable accesses. However, in current processors such accesses bypass the complete cache hierarchy, including the LLC. Contiguitas-HW piggybacks on existing hardware support for noncacheable accesses, but instead selectively activates noncacheable accesses while allowing caching in the LLC.

At a high level, one possible approach would be for Contiguitas-HW to set the noncacheable bit when a line is returned from the LLC to the private caches, notifying them to not cache the line. However, since noncacheable information is available a priori, at translation time, noncacheable requests issued by the core may follow a different and simpler path when traversing the cache hierarchy.

To this end, Contiguitas can integrate with existing noncacheable mechanisms as follows. When Contiguitas-HW issues a *BusRdX* it also notifies the cache agents in the private caches to install an entry in their miss status holding registers (MSHRs) corresponding to the source PPN, marking it as noncacheable. When the migration is complete Contiguitas-HW notifies the private caches to clear the entry. Finally, there is one special case that Contiguitas-HW needs to handle—a core that has not received an invalidation issues a request for the source page for the first time. In this case, Contiguitas-HW nacks the request and notifies the private caches as before. Then the cache agent will retry the cache access as noncacheable. This mechanism can further be implemented on top of upcoming hardware extensions [54] that will support efficient and extensible IPIs in hardware.

Cacheable Accesses. Noncacheable accesses resolve the challenge of maintaining coherence for a line that can be accessed by different mappings. However, with careful examination of the migration process, we identify that private caching can stay enabled as long as only a single mapping is used to cache a line at any given time. The benefit of this approach is twofold. First, lines under migration

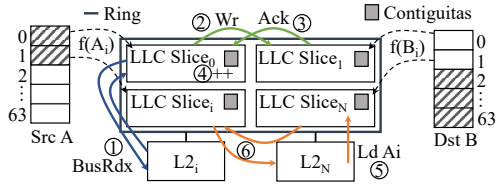

Figure 9: Contiguitas operations in a sliced last-level cache architecture.

can reside in private caches. Second, it eliminates the need to notify the private cache agents to handle migration pages as noncacheable.

To achieve this goal, Contiguitas-HW performs the following steps. First, when the OS notifies the Contiguitas-HW to start migrating a page, the hardware enables traffic redirection but does not start the copy process. Instead, the OS immediately modifies the page table entry to point to the destination page and starts invalidating each TLB. During this process, a page may be accessed with either the source or the destination mapping. If a request hits in the private caches, it is serviced without further action as in a regular cache hit. Otherwise, on a miss the Contiguitas-HW checks whether a line is currently stored in private caches with the opposite mapping of the request i.e., if a request is for the source mapping and the line is stored with the destination mapping, and vice versa. If the condition is true, the request invalidates any cached copy. Otherwise, the request is serviced regularly. This invariant allows the caching of lines under migration as only the source or the destination mapping is active in the private caches.

When the TLB invalidations are complete, the OS notifies the Contiguitas-HW to start the copy. At this point only the destination mapping is active at any given TLB. During the copy process, a request for the destination mapping might be in a modified state. In that case, the copying of that line is skipped as the destination already contains the most up-to-date data. Other operations are handled regularly. By the end of the migration the Contiguitas-HW notifies the OS as before.

Distributed Last-level Cache Slices. So far we have discussed the operations of Contiguitas-HW in the context of a monolithic last-level cache (LLC). In practice, the LLC usually consists of multiple slices that communicate over an interconnect such as a ring bus [52]. Figure 9 shows a high level overview of a sliced cache architecture. In this environment, a physical address is passed through a hash function f that spreads the cache lines of the page across different slices. As a result, a given source page A and destination page B, $f(A_i)$ and $f(B_i)$, may map each of their source and destination cache lines to different slices.

In this environment, Contiguitas-HW and its metadata table are replicated across each slice. In addition, Contiguitas-HW adds a copy of the slice selection hash function. This is needed because the hash function is usually located at the private L2 caches. The added hardware is cheap because such hash functions are a simple combination of logic gates e.g., XOR [116]. To use the *Src PPN* to initiate the copy of the lines, it first computes the slice selection function, $f(A_i)$ and checks if the address corresponds to the local slice.

If the line belongs to this slice, then Contiguitas-HW issues a *BusRdX* event as before, in Step ①. When the line is invalidated in

the private caches and the most recent version of the line is in the LLC slice, Contiguitas-HW uses the *Dst PPN*, $f(B_i)$ to identify the "home" slice of the destination line. If the home is the current slice then the copy is performed locally as before. Otherwise, Contiguitas-HW sends a Write request to the destination slice, shown in Step ②. After the write is complete, the destination slice responds in Step ③ with an acknowledgement. At this point, in Step ④ Contiguitas increments *Ptr*. If the line does not belong to this slice, Contiguitas-HW increments *Ptr* until it reaches a line that belongs to the local slice. This is correct because each local slice is only responsible for the lines within the slice.

Meanwhile, a request for the source page A_i may arrive at a slice, shown in Step ⑤. The slice responds in the following manner. First, it compares the address to *Ptr* to identify the migration status of the line as before. If the line has been migrated then Contiguitas-HW uses $f(B_i)$ to identify the destination slice. If the destination slice is the current one, then the operation continues locally as before. Otherwise, Contiguitas-HW does not need to perform the lookup in this slice. Instead, in Step ⑥ it forwards the request to the destination slice that then responds to the request. Overall, this process is similar to handling the case that a request arrives at an LLC slice but the line is dirty in a private cache.

The final piece of this process is how slices coordinate on when to perform their local copy procedures for a given page. With the above protocol in place and the natural parallelism provided by a distributed cache architecture, each slice could operate independently. This approach would significantly improve the performance of the copy operation. However, it would also add additional pressure to the interconnect during the copy operation. Instead, given the target operation of Contiguitas-HW of moving unmovable allocations, we opt for a simpler and less aggressive design. Specifically, after each slice completes its portion of the copies, it then sends a notification to the next slice to start. In the future, if an even lower latency page copy might be beneficial, leveraging slice parallelism might be more attractive.

Variable Buffer Sizes. Depending on the application characteristics it might be beneficial to support variable mapping sizes for device TLBs. Contiguitas-HW naturally supports this use case. When a mapping is composed of smaller independent translations, Contiguitas-HW can independently move each mapping one at at time. Alternatively, the metadata table of Contiguitas can be extended to maintain a *Size* field that defines the range of the translation. Depending on how the OS opts to map the allocation on the CPU side, the OS can perform TLB updates at a finer granularity than device TLB ones. Overall, Contiguitas-HW enables a large optimization space depending on the application characteristics.

Interface Contiguitas-HW supports two commands to interact with the OS. `Migrate (PPN_Src, PPN_Dst, Flag)` sets up a migration mapping by providing the source and destination physical page numbers. Based on the flag argument the mapping is simply installed or a migration is initiated based on the cacheable model of the microarchitecture. `Clear(PPN_Src)` clears the metadata table entry, ending a page migration.

Contiguitas-HW accepts commands from the OS via a work queue similarly to Intel's DSA [57]. The OS prepares a work descriptor in memory containing commands and parameters to Contiguitas-HW, and submits the descriptor to the work queue using the existing `ENQCMD` instruction [31]. The work descriptor contains a completion address that Contiguitas-HW writes to indicate to the OS that a work is completed.

4 EVALUATION METHODOLOGY

To evaluate Contiguitas, we follow a two-pronged approach. We evaluate the OS components of Contiguitas in Meta's production environment and further perform full-system simulations for the hardware components.

Production Environment. Unless otherwise noted, the hardware used for the production experiments is Intel Cooper Lake single-socket 64GB servers running Linux 5.12. We implement Contiguitas on top of Linux 5.12. In the rest of this section, we refer to the default kernel as *Linux*, our kernel implementation on top of Linux as *Contiguitas*, and the combination of our kernel implementation and hardware extensions as *Contiguitas-HW*. Since it is hard to run our production workloads on a simulator, we evaluate Contiguitas-HW with a page migration microbenchmark on top of NGINX [88], and memcached [35].

Production Workloads With Live Traffic. Our production evaluation leverages three major workloads at Meta, *Web*, *Cache A*, and *Cache B*. *Web* is a web server and one of the largest applications in deployment across Meta's fleet. *Cache A* and *Cache B* are the two largest in-memory caching services that handle traffic for most of the internal services at Meta. *Cache B* is a forked version of memcached. We leverage production-grade load-testing A/B infrastructure that has been developed internally at Meta to evaluate hardware selection and software optimizations. During our experiments, all workloads receive *real production traffic*.

Full-system Simulation. To evaluate the hardware components of Contiguitas we use full-system cycle-level simulations. We integrate QEMU [19] with SST [89] and DRAMSim3 [65]. We model a server architecture with 8 cores, 64GB of main memory. We run Linux 5.12 as the baseline and compare it with Contiguitas-HW. The architecture parameters are shown in Table 1. Each out-of-order core has private L1 and L2 caches, and a slice of the shared L3 cache. In addition, each core has private L1 and L2 TLBs and fully associative (FA) page walk caches. For Contiguitas's metadata table we use a fully associative structure with 16 entries per slice. To model TLB invalidations we intercept *INVLPG* instructions [52] issued by the OS and perform a traversal of the TLB hierarchy and page walk caches. Through measurements on a real processor, we find that the cost of the *INVLPG* instruction is substantially higher than the TLB round-trip and close to 250 cycles. We believe that it is due to the interactions of the instruction with the core pipeline. We model this behavior by adding a fixed penalty of 250 cycles, accounting for the cost of a full pipeline flush when an *INVLPG* instruction is issued.

Table 1: Architectural parameters.

Full-System Simulation Parameters	
Multicore Chip	8 4-issue OoO cores, 200-entry ROB, 2GHz
L1 Cache	32KB, 8-way, 2 cycles round trip (RT), 64B line
L1 TLB	64 entries, 4-way, 2 cycles RT
L2 TLB	1536 entries, 16-way, 12 cycles RT
Page Walk Cache	3 levels, 32 entries per level, FA, 2 cycles
L2 cache	256KB, 8-way, 14 cycles RT
L3 cache	2MB slice, 16-way, 40 cycles RT
Contiguitas-HW	16 entries, FA, 1 cycle
Main memory	64GB, DDR4 3200, 16 Banks, 1GHz
OS	Linux 5.12

5 EVALUATION

Our evaluation attempts to answer the following questions:

(1) Does Contiguitas improve application's end-to-end performance?
(2) What is Contiguitas's impact on unmovable allocations?
(3) What is Contiguitas's impact on memory contiguity?
(4) Does page migration in Contiguitas-HW scale with the number of cores?
(5) How do Contiguitas-HW page migrations affect page availability and application performance?
(6) What are the hardware resource requirements of Contiguitas-HW?

5.1 End-to-end Performance Impact

We evaluate the performance of three major workloads at Meta, *Web*, *Cache A*, and *Cache B*. The performance metric is requests per second under certain latency SLAs based on the characteristics of each workload. In all cases, THP is set to "always". Additionally, Web attempts to allocate 2MB and 1GB huge pages directly.

We consider two setups, *Full Fragmentation* and *Partial Fragmentation*. *Full Fragmentation* represents the case where a workload lands on a server whose memory is already fully fragmented. As we showed in Section 2.4, 23% of production servers are fully fragmented. To represent this scenario, we run a fragmentation process to fragment the server fully before the workload is deployed.

Partial Fragmentation represents the case where a workload lands on a partially fragmented server that is representative of the majority of servers at Meta. As each workload fragments the address space at different levels, we first run the same workload to fragment the server and then restart the workload before our experiment. This behavior is common in production due to frequent code deployments.

Figure 10 shows the results. Contiguitas produces the same result under *Full Fragmentation* and *Partial Fragmentation* and hence we show one bar. In "Linux Partial" and "Linux Full", dynamically allocating 1GB HugeTLB pages always fails due the lack of contiguity. For Web, we show a stacked bar for Contiguitas to highlight the gain with 1 GB pages. Overall, Contiguitas delivers substantial performance improvements between 2% and 18% across major workloads at Meta.

Looking at Web, compared to Linux in the fully fragmented case, Contiguitas achieves 18% higher performance. Compared to Linux in the partial fragmented case, Contiguitas delivers a performance gain of 9%. Notably, looking at the breakdown of Contiguitas we see that substantial gains are due to Contiguitas being able to *dynamically* allocate 1 GB HugeTLB pages. Diving deeper, we find that Linux with partial fragmentation was able to allocate 14 GB worth

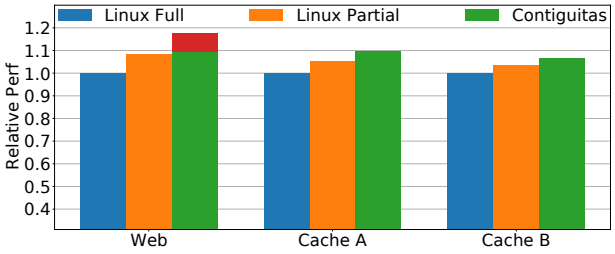

Figure 10: End-to-end performance over Meta's production workloads. The red bar of Contiguitas for Web shows the performance gains from 1GB pages.

of 2 MB huge pages compared to 20 GB allocated by Contiguitas. For 1 GB huge pages, Contiguitas allocated 4GB in total whereas Linux allocated none. Note that, in Contiguitas, 1 GB pages provided a major performance win of 7.5%, which highlights the importance of bigger contiguity beyond 2 MB. Finally, we compared Contiguitas and Linux when both were forced to use only 4 KB pages. While the results are not shown in the figure, they had identical performance, showing that Contiguitas introduces no measurable overhead even if it finds no opportunities to allocate huge pages in the worst case.

5.2 Unmovable Allocations and Memory Contiguity.

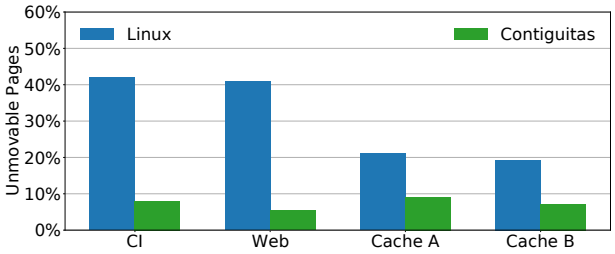

Figure 11: Unmovable 2 MB pages for Meta's production workloads.

Unmovable allocations. To assess the impact of Contiguitas on unmovable allocations, we compare the percentage of memory that becomes unmovable for different workloads at Meta. We further include Meta's continuous-integration (CI) workload that executes various build and test jobs. For this experiment, we start our workloads on freshly booted servers and then profile the servers once every 15 minutes.

Figure 11 shows the percentages of 2 MB unmovable pages. We observe that for all workloads we consider, unmovable memory increases drastically within the first hour and then plateaus. Therefore, we show the percentages at their stable state. As we discussed in Section 2.4, given the long uptime of servers, fragmentation is ubiquitous across the fleet. Under Linux, the unmovable memory is between 19% to 42%, with an average of 31%. In contrast, Contiguitas has the highest value at 9%, with an average of 7%. This demonstrates that Contiguitas's confinement strategy successfully restricts the spread of unmovable 2MB pages and keeps the unmovable region small.

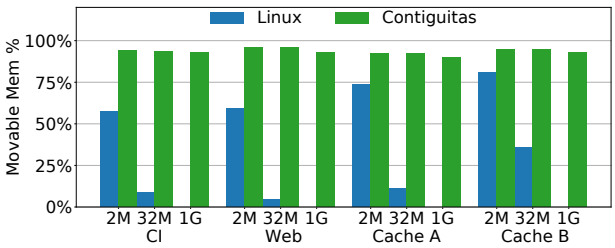

Figure 12: Potential memory contiguity as a percentage of total memory.

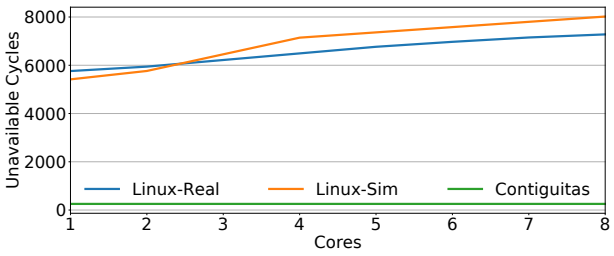

Figure 13: Page unavailable cycles during migration.

To understand the internals of the unmovable region, we characterize its fragmentation. Specifically, we scan the unmovable region and count the number of unmovable pages within each 2 MB block. Note that, in the unmovable region a page is either unmovable or free. We find that 22% of the pages in a typical 2 MB block in the unmovable region are free. This internal fragmentation of the unmovable region further motivates the need for Contiguitas-HW, especially as the number of unmovable allocations increases (Section 2.5).

Potential memory contiguity. To quantify the impact of Contiguitas on memory contiguity, we compare each workload's steady state under Linux and Contiguitas. Specifically, we quantify the contiguous regions that can be formed if we *hypothetically* run a perfect software compaction in order to service allocation requests of 2 MB, 32 MB, and 1 GB. Figure 12 shows the results. With Linux we see that some 2 MB allocation are possible given that less than half the memory is composed of unmovable 2MB pages as we discussed above. However, Linux struggles as we search for larger contiguous regions, and fails to find even a single 1 GB page. On the other hand, Contiguitas, by design, isolates the unmovable region and hence the whole movable region can potentially be used after compaction for large contiguous allocations, even 1 GB pages as we showed in Section 5.1.

5.3 Contiguitas-HW Characterization.

Scalability. To evaluate the scalability of Contiguitas-HW with respect to the number of cores we use full system simulations. Specifically, we develop a micro-benchmark that triggers page migrations in software and Contiguitas-HW. We simulate all the operations of Contiguitas-HW and instrument Linux to track the page migration procedure, including TLB shootdowns and page copy operations. We then measure the number of cycles a page is unavailable during migration from the perspective of a memory operation to that page. Figure 13 shows the results. The measurement with one core means that we perform a TLB shootdown on one remote core. *Linux-Sim* shows the result of a simulated system, which is compared to the result of a real system, denoted *Linux-Real*. We see that the two systems match well (-6% to +10% difference), which confirms the accuracy of the simulation.

From the results we see that as the number of cores/TLBs involved in the TLB shootdown increases, the amount of time during which access to the page is blocked increases linearly. The cost of the page copy remains the same, at approximately 1,300 cycles. Finally, we plot the result from *Contiguitas-HW*. By design, the cost of Contiguitas-HW is constant and equal to a local TLB invalidation, because its TLB shootdowns do not require inter-processor interrupts and each TLB can be locally invalidated without coordination or synchronous acknowledgements. The cost of a 4 KB page migration in Contiguitas-HW is close to 2us in our microbenchmark. Although the absolute migration time will vary based on the cache behavior of the page, access to the page is never blocked during a migrations.

Performance. We evaluate with full system simulations the performance implications of Contiguitas-HW on two open source applications NGINX and Memcached. We select these workloads to cover both ends of the spectrum. NGINX is insensitive to huge pages and it makes use of networking buffers. In this scenario, our goal is to move umovable pages that will benefit other applications without interfering with the application that makes use of unmovable allocations. Memcached is a close proxy to our Cache B workload, which is a fork of memcached; it is sensitive to huge page availability and also relies on networking buffers.

In both cases we consider an aggressively disadvantageous scenario where Contiguitas-HW moves unmovable networking buffers only of the application while the application serves requests at peak throughput without any available slack. We consider two rates: (i) *Regular* is the expected rate of unmovable page movement set at 100/second based on the ratio of unmovable and movable pages, (ii) *Very High* which is the rate if we were to move unmovable pages at the highest rate we observed our production systems to move movable pages, set at 1000/second. We evaluate both workloads without huge pages in order to understand any potential negative performance impact by unmovable page movement without any performance gains from huge pages.

At *Regular* rate Contiguitas-HW with noncacheable and cacheable accesses do not have an impact on application performance. Even at *Very High* rate, which would be unwarranted for a real environment, Contiguitas-HW with noncacheable accesses only shows a marginal overhead of 0.2% for NGINX and 0.3% for memcached. The Contiguitas-HW with cacheable accesses does not have an impact on application performance even at this rate. This is because Contiguitas-HW is able to very efficiently redirect traffic and perform copy operations in the background.

When combined with the benefits of contiguity and 2MB huge pages, memcached performance improves by 7%. Overall, Contiguitas-HW does not negatively impact applications that do not benefit from contiguity while improving contiguity for those that do. In addition, Contiguitas-HW successfully achieves its goal to drastically reduce the unmovable region. Specifically, it enables the migration of the majority of the unmovable pages and further enables

the defragmentation of the unmovable region that can potentially unblock 22% of the region's memory (Section 5.2).

Sizing and Hardware Requirements. To size the metadata table of Contiguitas-HW we consider the following parameters. First, Contiguitas-HW enables local TLB invalidations that can occur whenever the kernel naturally executes on top of a core. Hence, we need to take into account the amount of time a mapping needs to be maintained. To this end we consider the rate of system calls and context switches on a core that is observed to be 40K-100K per second in production. This provides a window of at least 25us for an invalidation to take place. Accounting conservatively for 5us for a 4KB page copying to take place this provides us with 30us per migration. As a result, a single entry already provides a very high theoretical number of migrations/second.

To facilitate concurrent migrations, we size the metadata table to store 16 entries as a fully associative structure. Requests across slices are buffered in existing slice queues that handle regular cache requests. Contiguitas-HW augments messages to pass additional information e.g., noncacheable. However, it does not require additional buses as requests and responses are flowing in existing paths between L3 slices and the L2 caches. Our Cacti [13] analysis for a $22nm$ technology node shows that the area for Contiguitas-HW per slice is $0.0038mm^2$, the energy per access is $0.0017nj$, and the leakage power is $0.64mW$. We conservatively account for 2-cycle access time to the metadata table. Compared to the size of a core, the cost of Contiguitas-HW is only 0.014%. Overall, the hardware cost of Contiguitas-HW is negligible.

6 DISCUSSION

As server memory capacity increases drastically, resolving excessive memory management overheads will require a fundamental rebuild of the operating system and hardware layers. Contiguitas tames unmovable allocations and introduces a reliable source of physical memory contiguity that can enable a plethora of prior [38, 40, 43, 44, 48, 64, 67, 68, 72, 77, 78, 82, 83, 95, 98, 99, 108, 114, 115] and future works that aim to tackle the address translation overhead. In the near term, we expect that 2 MB huge pages can become a primary choice for memory intensive applications and curb the address translation overhead. Finally, existing 1 GB huge pages can provide major performance benefits for certain applications when carefully deployed, however we believe that additional support and research is needed as the substantial increase in page size can exacerbate the severity of data movement overheads and memory bloating.

7 RELATED WORK

TLB efficiency has been a major target of prior work [15, 23, 29, 60, 60, 69, 72, 75, 79, 81, 82, 94, 97, 108, 114]. Works have focused on coalesced [81], range-based [60], and part of memory TLB designs [70, 92]. Contiguitas can boost their efficiency by providing ample contiguity.

Huge page support has received significant attention in the past [39, 48, 64, 67, 68, 77, 78, 87, 102, 113, 114]. Other efforts have focused on upstreaming 1GB THP support [113]. As we showed in Section 2.2, 1GB huge pages can be effective in reducing the address translation cost. Work on capabilities [111] can reduce address translation overheads; the approach of Contiguitas could be useful in such environments in order to efficiently manage DMA and other sources of unmovable pages. More recent proposals have also focused on improving the efficiency of userspace allocators to better utilize huge pages [48, 67, 68]. Contiguitas can further boost the efficiency of userspace allocators as they rely on the OS to provide contiguity.

Another body of work has focused on improving TLB shootdown operations [7, 8, 12, 17, 63, 63, 90, 109]. For example, UNITD [90] proposes TLBs to be included in the regular cache coherence protocol in order to reduce their overhead. LATR [63] proposes asynchronous TLB invalidation's for operations that are amenable to lazy TLB shootdowns but this mechanism is not applicable to unmovable pages because such mappings cannot afford to become unavailable. Overall, the primary focus of these works is to perform fast TLB shootdown operations so their frequency can be increased [12]. Instead, the goal of Contiguitas is to enable transparent migration of unmovable pages while in use. As a result, Contiguitas arrives at a vastly different design point that requires minimal hardware changes in the last-level cache.

8 CONCLUSION

This paper presented Contiguitas, a holistic solution across the operating system and hardware that reduces memory fragmentation due to unmovable allocations. Our evaluation at Meta's datacenters and full-system simulations showed that Contiguitas is a generic, production-ready solution to provide ample memory contiguity for not only the current demand but also a future where the optimal page size will continue to grow alongside ever-increasing memory capacities.

ACKNOWLEDGMENTS

This work was funded in part by NSF grants CNS-2239311, CNS-2107307, CCF-2217016, by Intel Corp under the SRS award, and a Meta Faculty Award. We thank David Koufaty, Nastaran Hajinazar, Alex Daglis and the anonymous reviewers for all of their valuable feedback.

REFERENCES

[1] Keith Adams and Ole Agesen. 2006. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.

[2] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. 2012. Revisiting Hardware-assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12)*. Portland, Oregon.

[3] Hana Alam, Tianhao Zheng, Mattan Erez, and Yoav Etsion. 2017. Do It Yourself Virtual Memory Translation. In *the Proceedings of ISCA*. Toronto, Canada, 1—12. https://doi.org/10.1145/3079856.3080209

[4] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA'20)*.

[5] AMD. 2022. AMD I/O Virtualization Technology (IOMMU) Specification. https://www.amd.com/en/support/tech-docs/amd-io-virtualization-technology-iommu-specification.

[6] AMD. 2022. Invalidate TLB Entry(s) with Broadcast. https://www.amd.com/system/files/TechDocs/24594.pdf.

[7] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association,

USA, 27–39.
[8] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't Shoot down TLB Shootdowns!. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 35, 14 pages. https://doi.org/10.1145/3342195.3387518
[9] ArangoDB. 2018. Transparent Huge Pages Warning. https://github.com/arangodb/arangodb/blob/2b84348b7789893878ebd0c8b552dc20416c98f0/lib/ApplicationFeatures/EnvironmentFeature.cpp.
[10] ARM. 2023. Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.
[11] ARM. 2023. TLBI (A64). https://developer.arm.com/documentation/dui0801/l/A64-General-Instructions/TLBI--A64-.
[12] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 273–287. https://doi.org/10.1109/PACT.2017.38
[13] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization* 14, 2, Article 14 (June 2017), 14:1–14:25 pages.
[14] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2010. Translation Caching: Skip, Don't Walk (the Page Table). In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*.
[15] Thomas W. Barr, Alan L. Cox, and Scott Rixner. 2011. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*.
[16] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*.
[17] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579
[18] N. Beckmann, P. Tsai, and D. Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 538–550.
[19] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 10–5555.
[20] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*.
[21] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*.
[22] Abhishek Bhattacharjee. 2017. Translation-Triggered Prefetching. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.
[23] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. 2011. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.
[24] Abhishek Bhattacharjee and Margaret Martonosi. 2010. Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*.
[25] Travis Campbell. 2015. Transparent Huge Pages on Hadoop makes me sad. https://www.ghostar.org/2015/02/transparent-huge-pages-on-hadoop-makes-me-sad.
[26] Cloudera. 2021. Transparent Huge Pages Warning. https://www.cloudera.com/documentation/enterprise/latest/topics/cdh_admin_performance.html.
[27] Compute Express Link Consortium. 2022. CXL 3.0 White Paper. https://www.computeexpresslink.org/_files/ugd/0c1418_a8713008916044ae9604405d10a7773b.pdf.
[28] Jonathan Corbet. 2010. Memory compaction. https://lwn.net/Articles/368869/.
[29] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.
[30] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.
[31] The Linux Kernel Documentation. 2023. Shared Virtual Addressing (SVA) with ENQCMD. https://docs.kernel.org/x86/sva.html.
[32] Cort Dougan, Paul Mackerras, and Victor Yodaiken. 1999. Optimizing the Idle Task and Other MMU Tricks. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*.
[33] Stephane Eranian and David Mosberger. 2000. *The Linux/ia64 Project: Kernel Design and Status Update.* Technical Report HPL-2000-85. HP Labs.
[34] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone
[35] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (2004).
[36] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. Cambridge, United Kingdom.
[37] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*.
[38] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*.
[39] Mel Gorman and Patrick Healy. 2008. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management (ISMM '08)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/1375634.1375641
[40] Mel Gorman and Patrick Healy. 2010. Performance Characteristics of Explicit Superpage Support. In *Proceedings of the 2010 International Conference on Computer Architecture (ISCA'10)*.
[41] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. 2005. Itanium — A System Implementor's Tale. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)*.
[42] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 202–215. https://doi.org/10.1145/2934872.2934908
[43] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. 2015. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th ACM International Conference on Virtual Execution Environments (VEE'15)*.
[44] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. 2021. Rebooting Virtual Memory with Midgard. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 512–525. https://doi.org/10.1109/ISCA52012.2021.00047
[45] Swapnil Haria, Mark D. Hill, and Michael M. Swift. 2018. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. Williamsburg, VA, USA.
[46] Red Hat. 2015. Red Hat Enterprise Linux Network Performance Tuning Guide. https://access.redhat.com/sites/default/files/attachments/20150325_network_performance_tuning.pdf.
[47] Jerry Huck and Jim Hays. 1993. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*.
[48] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 257–273. https://www.usenix.org/conference/osdi21/presentation/hunter
[49] IBM. 2005. PowerPC Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors. https://wiki.alcf.anl.gov/images/f/fb/PowerPC_-_Assembly_-_IBM_Programming_Environment_2.3.pdf.
[50] Intel. 2010. Itanium Architecture Software Developer's Manual (Volume 2). https://www.intel.com/content/www/us/en/products/docs/processors/itanium/itanium-architecture-vol-1-2-3-4-reference-set-manual.html.
[51] Intel. 2015. 5-Level Paging and 5-Level EPT (White Paper). https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf.
[52] Intel. 2019. 64 and IA-32 Architectures Software Developer's Manual.
[53] Intel. 2021. Intel Trusted Domain Extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf.

[54] Intel. 2021. Remote Action Request White Paper. https://www.intel.com/content/dam/develop/external/us/en/documents/341431-remote-action-request-white-paper.pdf.

[55] Intel. 2022. Intel Virtualization Technology for Directed I/O. https://edc.intel.com/content/www/xl/es/publications/specification-nuc12dcm-nuc12edb/intel-virtualization-technology-for-directed-i-o/.

[56] Intel. 2022. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[57] Intel. 2023. Introducing the Intel Data Streaming Accelerator. https://01.org/blogs/2019/introducing-intel-data-streaming-accelerator.

[58] Bruce L. Jacob and Trevor N. Mudge. 1998. A Look at Several Memory Management Units, TLB-refill Mechanisms, and Page Table Organizations. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*.

[59] Joefon Jann, Paul Mackerras, John Ludden, Michael Gschwind, Wade Ouren, Stuart Jacobs, Brian F. Veale, and David Edelsohn. 2018. IBM POWER9 system software. *IBM Journal of Research and Development* 62, 4/5 (June 2018).

[60] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. 2015. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*.

[61] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. 2016. Energy-Efficient Address Translation. In *Proceedings of 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*.

[62] Samuel T. King, George W. Dunlap, and Peter M. Chen. 2003. Operating System Support for Virtual Machines. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC'03)*.

[63] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy Translation Coherence. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 651–664. https://doi.org/10.1145/3173162.3173198

[64] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[65] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. 2020. DRAMsim3: a cycle-accurate, thermal-capable DRAM simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109.

[66] Matt Loughran. 2015. How Okta Chased Down Severe System CPU Contention in MySQL. https://developer.okta.com/blog/2015/05/22/tcmalloc.

[67] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[68] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S. McKinley, and Paul Turner. 2021. 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* (Virtual, Canada) *(ISMM 2021)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/3459898.3463905

[69] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. 2015. rIOMMU: Efficient IOMMU for I/O Devices that Employ Ring Buffers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 355–368. https://doi.org/10.1145/2694344.2694355

[70] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. 2017. CSALT: Context Switch Aware Large TLB. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*. Cambridge, MA, USA.

[71] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1023–1036. https://doi.org/10.1145/3352460.3358294

[72] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*.

[73] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 399–413. https://doi.org/10.1145/3341301.3359657

[74] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. 2014. *The design and implementation of the FreeBSD operating system.* Pearson Education.

[75] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jimenez. 2020. CHiRP: Control-Flow History Reuse Prediction. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*.

[76] MongoDB. 2018. Transparent Huge Pages Warning. https://github.com/mongodb/mongo/blob/eeea7c2f80bdaf49a197a1c8149d7bc6cbc9395e/src/mongo/db/startup_warnings_mongod.cpp.

[77] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. 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 (ASPLOS'19)*.

[78] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*.

[79] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. 2017. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*.

[80] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. 2015. Utilizing the IOMMU Scalably. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 549–562. https://www.usenix.org/conference/atc15/technical-session/presentation/peleg

[81] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H. Loh. 2014. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*.

[82] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. 2012. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*.

[83] Binh Pham, Ján Veselỳ, Gabriel H. Loh, and Abhishek Bhattacharjee. 2015. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have it Both Ways?. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*. Waikiki, Hawaii, USA.

[84] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The benefits of general-purpose on-NIC memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland, 1130–1147.

[85] DPDK Project. 2019. Memory in DPDK, Part 1: General Concepts. https://www.dpdk.org/blog/2019/08/21/memory-in-dpdk-part-1-general-concepts/.

[86] Moinuddin K. Qureshi. 2018. CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*. IEEE Press, Fukuoka, Japan, 775–787. https://doi.org/10.1109/MICRO.2018.00068

[87] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. 2021. Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1106–1120. https://doi.org/10.1145/3466752.3480062

[88] Will Reese. 2008. Nginx: The High-Performance Web Server and Reverse Proxy. *Linux J.* (2008).

[89] Arun F Rodrigues, Gwendolyn Renae Voskuilen, Simon David Hammond, and Karl Scott Hemmert. 2016. *Structural Simulation Toolkit (SST)*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[90] Bogdan F. Romanescu, Alvin R. Lebeck, Daniel J. Sorin, and Anne Bracy. 2010. UNified Instruction/Translation/Data (UNITD) coherence: One protocol to rule them all. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416643

[91] Mark E Russinovich, David A Solomon, and Alex Ionescu. 2012. *Windows internals, part 2.* Pearson Education.

[92] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. 2017. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. Toronto, ON, Canada.

[93] Samsung. 2019. Samsung Electronics Introduces Industry's First 512GB CXL Memory Module. https://news.samsung.com/global/samsung-electronics-introduces-industrys-first-512gb-cxl-memory-module.

[94] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. 2020. BabelFish: Fusing Address Translations for Containers. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA) (ISCA 2020)*. https://doi.org/10.1109/ISCA45697.2020.00049

[95] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2020. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*.

[96] J. E. Smith and Ravi Nair. 2005. The Architecture of Virtual Machines. *IEEE Computer* 38, 5 (2005), 32–38. https://doi.org/10.1109/MC.2005.173

[97] Shekhar Srikantaiah and Mahmut Kandemir. 2010. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*.

[98] Jovan Stojkovic, Namrata Mantri, Dimitrios Skarlatos, Tianyin Xu, and Josep Torrellas. 2023. Memory-Efficient Hashed Page Tables. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1221–1235. https://doi.org/10.1109/HPCA56546.2023.10071061

[99] Jovan Stojkovic, Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. 2022. Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. https://doi.org/10.1145/3503222.3507720

[100] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VI)*. Association for Computing Machinery, New York, NY, USA, 171–182. https://doi.org/10.1145/195473.195531

[101] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*.

[102] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. 1992. Tradeoffs in Supporting Two Page Sizes. In *19th International Symposium on Computer Architecture (ISCA'92)*.

[103] Mellanox Technologies. 2019. Introducing 200G HDR InfiniBand Solutions. https://network.nvidia.com/files/doc-2020/wp-introducing-200g-hdr-infiniband-solutions.pdf.

[104] Mellanox Technologies. 2020. Mellanox Adapters Programmer's Reference Manual. https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf.

[105] The Linux Kernel Archives. 2019. Transparent Hugepage Support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt.

[106] The Linux Kernel Archives. 2023. HugeTLB Pages. https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html.

[107] The Linux Kernel Archives. 2023. x86 virtual memory map. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt.

[108] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. 2021. Exploiting Page Table Locality for Agile TLB Prefetching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 85–98. https://doi.org/10.1109/ISCA52012.2021.00016

[109] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 340–349. https://doi.org/10.1109/PACT.2011.65

[110] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. https://doi.org/10.1145/3503222.3507731

[111] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)* (Minneapolis, Minnesota, USA).

[112] www.7-cpu.com. 2023. Intel Skylake Timing. https://www.7-cpu.com/cpu/Skylake.html.

[113] Zi Yan. 2020. 1GB THP support on x86_64. https://lwn.net/Articles/830435/.

[114] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA'19)*. Phoenix, AZ, USA.

[115] Idan Yaniv and Dan Tsafrir. 2016. Hash, Don't Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS'16)*.

[116] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel Last-Level Cache. *IACR Cryptology ePrint Archive* 2015 (2015), 905. https://eprint.iacr.org/2015/905

[117] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2785956.2787484