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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '23, June 17–21, 2023, Orlando, FL, USA.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
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.

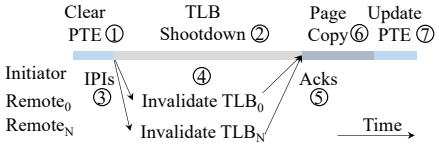

Figure 1: Page migration procedure.

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

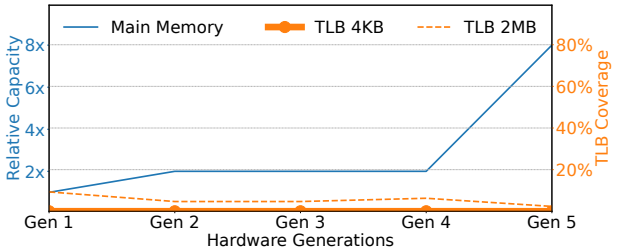

Figure 2: Memory and TLB coverage of computing hardware across generations.

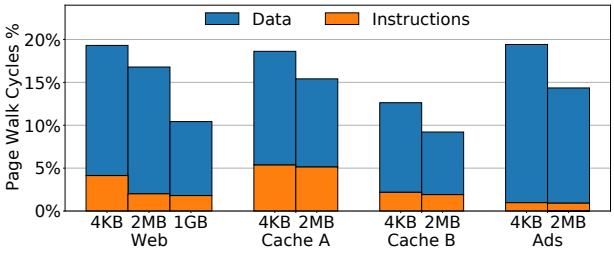

Figure 3: Percentage of cycles lost due to page walks.

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

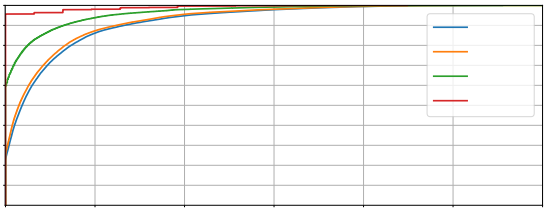

Figure 4: Contiguity availability as the percentage of free memory.

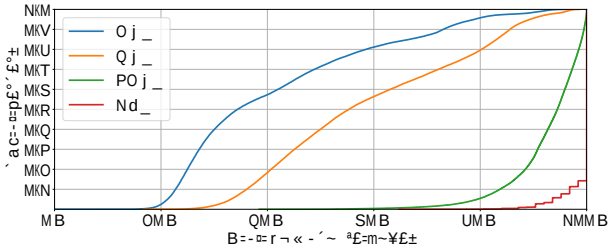

Figure 5: Distribution of unmovable pages in contiguous regions of 2MB, 4MB, 32MB, and 1GB.

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.

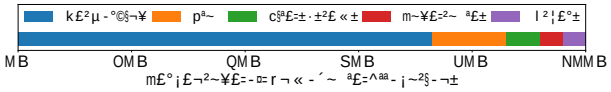

Figure 6: Sources of unmovable allocations.

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