# MEGA: Overcoming Traditional Problems with OS Huge Page Management

Theodore Michailidis
University of Athens
Athens, Greece
tmichailidis@di.uoa.gr

Alex Delis
University of Athens
Athens, Greece
ad@di.uoa.gr

Mema Roussopoulos
University of Athens
Athens, Greece
mema@di.uoa.gr

## ABSTRACT

Modern computer systems now feature memory banks whose aggregate size ranges from tens to hundreds of GBs. In this context, contemporary workloads can and do often consume vast amounts of main memory. This upsurge in memory consumption routinely results in increased virtual-to-physical address translations, and consequently and more importantly, more translation misses. Both of these aspects collectively do hamper the performance of workload execution. A solution aimed at dramatically reducing the number of address translation misses has been to provide hardware support for pages with bigger sizes, termed *huge pages*. In this paper, we empirically demonstrate the benefits and drawbacks of using such huge pages. In particular, we show that it is essential for modern OS to refine their software mechanisms to more effectively manage huge pages. Based on our empirical observations, we propose and implement MEGA, a framework for huge page support for the `Linux` kernel. MEGA deploys basic tracking mechanisms and a novel memory compaction algorithm that jointly provide for the effective management of huge pages. We experimentally evaluate MEGA using an array of both synthetic and real workloads and demonstrate that our framework tackles known problems associated with huge pages including increased page fault latency, memory bloating as well as memory fragmentation, while at the same time it delivers all huge pages benefits.

## CCS CONCEPTS

• **Software and its engineering → Operating systems**; **Memory management**; **Allocation / deallocation strategies**;


*SYSTOR '19, June 3–5, 2019, Haifa, Israel*

ACM ISBN 978-1-4503-6749-3/19/06…$15.00
https://doi.org/10.1145/3319647.3325839

## KEYWORDS

Huge pages, Address Translation, Memory Compaction

**ACM Reference Format:**
Theodore Michailidis, Alex Delis, and Mema Roussopoulos. 2019. MEGA: Overcoming Traditional Problems with OS Huge Page Management. In *The 12th ACM International Systems and Storage Conference (SYSTOR '19), June 3–5, 2019, Haifa, Israel.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3319647.3325839

## 1 INTRODUCTION

Computer memory capacities have been increasing significantly over the past decades, leading to the development of memory hungry workloads that consume vast amounts of main memory. This upsurge in memory consumption routinely results in increased virtual-to-physical address translations and consequently, and more importantly, an increase in TLB translation misses. Increased TLB translation misses can cause execution time overheads of up to 50% [2, 4, 13]. Some proposed solutions to this problem, such as increasing the TLB size or adding more page table levels simply increase average and worst-case translations costs. However, one proposed solution, dating back to the 1990s aims to dramatically reduce the number of address translation misses by providing hardware support for pages with bigger sizes, known as *huge pages*. Depending on the architecture, multiple huge page sizes exist, with the most common being 2MB or 1GB. As a result of hardware changes to add huge pages, software techniques in modern OS have been developed to exploit huge pages and their benefits.

Huge pages can significantly reduce address translations, and consequently address translation misses in the TLB. Unfortunately, due to poor huge page management in modern operating systems, huge pages also bring a number of problems such as increased page fault latency, increased memory fragmentation and memory bloating, and high CPU usage. These problems, combined with the fact that for many years TLBs provided limited entries for huge pages, led multiple software vendors to recommend disabling huge page support [11, 20, 24, 33, 34]. However, in recent years, hardware vendors have dramatically increased the number of TLB entries provided for huge pages, making them a more attractive

option again. Thus, it is imperative for the OS community to design new sophisticated algorithms that allow operating systems to efficiently exploit huge pages, while minimizing as much as possible their associated shortcomings.

In this paper, we empirically demonstrate the benefits and drawbacks of using huge pages. Based on our empirical observations, we design, implement, and evaluate MEGA: **M**anaging **E**fficiently Hu**g**e P**a**ges[1], a memory management framework for huge page support for the Linux kernel. The key insight of MEGA is to provide huge pages only when they are needed and when their use does not impair system and workload performance. We achieve this by tracking the locality of virtual-to-physical mappings and their utilization throughout execution. We introduce a novel compaction algorithm that distinguishes physical memory blocks based on their utilization and age of last virtual-to-physical mapping.

We experimentally evaluate MEGA using an array of both synthetic and real workloads and demonstrate that compared to previous work, MEGA tackles more effectively the problems associated with the use of huge pages. Specifically, compared with the mainstream Linux kernel with huge page support, MEGA achieves an order of magnitude lower page fault tail latencies. We show that both Linux and Ingens, a recent, state-of-the-art huge page management framework, suffer from gratuitous use of huge pages due to aggressive policies. On the contrary, MEGA avoids this, via intelligent selection of huge page placement. Moreover, our memory compaction algorithm, based on a novel cost-benefit approach, consistently achieves up to 2X the amount of available free memory compared with Ingens, thus greatly facilitating memory allocation.

## 2 BACKGROUND

In this section, we briefly present how the Linux kernel currently handles some aspects of huge pages and compaction and then discuss its benefits and drawbacks.

### 2.1 Hugepage/Superpage OS support

Initial huge page support in Linux was through hugetlbfs [35]; users were able to use huge pages by explicitly requesting them in their application either via the mmap system call with the MAP_HUGETLB flag or the shmget system call with the SHM_HUGETLB flag [21] [22]. However, this policy burdens the developer who must explicitly state how and when huge pages must be used, and also does not cope well with non-database workloads [10]. Despite these disadvantages, this approach is currently followed by the macOS and Windows operating systems. Specifically, in macOS, the user asks for a huge page by using the VM_FLAGS_SUPERPAGE_* flag in the mmap system call [23]. Likewise, in Windows, the user must use the MEM_LARGE_PAGES as allocation type in the VirtualAlloc function [8], to allocate non-pageable memory, using large pages [30].

To avoid the drawbacks of hugetlbfs, the Transparent Huge Pages (THP) feature was developed in Linux 2.6.38 [28], released in March 2011. With THP, the handling of huge pages is done transparently by the kernel, without user involvement; however, users can force the kernel to use huge pages by using mmap with the MAP_POPULATE flag. Initially, the kernel uses base pages, i.e. 4KB pages, to satisfy memory requests from the user level. We say the kernel **promotes** a sequence of 512 properly aligned base pages to a huge page (and **demotes** a huge page into 512 base pages). Currently, Linux's policy is to promote even a single base page, that was faulted, to a huge page. Correspondingly, it demotes a huge page immediately, if any portion of its memory is freed.

FreeBSD also uses a transparent model to handle superpages, which is based on reservation-based allocations with variable sized superpages and incremental promotions/demotions [19].

### 2.2 Memory compaction

Linux divides physical memory into four memory zones [6]; we briefly introduce how they are used and which subset of memory they occupy in the x86_64 architecture. **ZONE_DMA**: It is used for Direct Memory Access (DMA), primarily by Industry Standard Architecture (ISA) devices which are limited to 24-bit addresses, and contains the first 16MB of memory. **ZONE_DMA32**: It exists only in the x86_64 architecture and is primarily used by hardware that uses 32-bit DMA, and contains physical pages that lie in [16MB, 4GB]. **ZONE_NORMAL**: It contains normal addressable pages and, in the 64-bit kernel, it contains addresses that lie in [4GB, end of memory]. In x86_64, the kernel tries to satisfy user-level memory requests from ZONE_NORMAL, and if there is no available memory, it tries to allocate memory from ZONE_DMA32.

Memory compaction is the process that the kernel follows to defragment the memory. The current kernel memory compaction algorithm [12] performs the following steps in each zone:

- Initially, two scanners are used; the first one (the **migration** scanner) starts from the beginning of the zone and keeps track of allocated pages, while the second (the **free** scanner) starts from the end of the zone and keeps track of free pages. The scanners continue until they meet.
- After the scanners meet, the page migration algorithm is invoked. Initially, for each group of pages that will be migrated, a corresponding chunk of free pages should be found. After that, the kernel **isolates** both the pages

[1] Also, from the Ancient Greek word μέγα, which means large

that will be migrated and the free pages. This isolation involves acquiring heavily contended locks and inserting the pages to be moved and the free pages into two distinct lists. Right after, the algorithm checks if the former are movable and removes from the list the ones that are not.
- Finally, the kernel migrates the pages to the corresponding available space.

The goal of this compaction algorithm is to separate free and allocated memory by grouping free pages at the start of the adress space, while organizing allocated pages close to the end of the address space.

The Linux kernel 4.6 release introduced a set of background kernel threads, kcompactd. There is one kernel thread per CPU and their purpose is to compact memory when it is needed. Specifically, kcompactd starts compacting when one of the following happens: (1) a multipage, i.e. a batch of multiple contiguous pages, allocation fails, (2) a set of base pages is going to be promoted to a huge page and there is no 2MB of contiguous memory available, (3) khugepaged tries to compact memory asynchronously, to satisfy future THP page faults, (4) it is triggered manually by writing a value in /proc/sys/vm/compact_memory.

Linux uses a daemon thread, kswapd [27], which is responsible for swapping out modified pages to the swap file, to increase the number of free available pages. kcompactd is called after kswapd has woken up and tried to reclaim memory. Whether kcompactd will skip the compaction or not, depends on the value of the fragmentation index [9], an index that represents how fragmented the memory is. The index is based on the total available memory and the available number of contiguous 2MB chunks of memory. It skips compaction if the fragmentation index is not -1 and lower than 500; in the former case, the requested allocation can be satisfied, in the latter, the problem is that there is no sufficient amount of memory. Finally, the kernel tracks and provides some statistics about compaction. **(1)**#pages scanned for migration, **(2)**#pages scanned in search of free pages, **(3)**#isolated pages, i.e. both free and movable base pages that were isolated for migration, **(4)**#successfully migrated pages, and **(5)**#failed migrated pages.

## 2.3 Why use Huge Pages

Huge pages can provide a number of benefits. **First**, studies have shown that TLB misses account for up to 50% of the total execution time [13]. By grouping multiple 4KB pages into one 2MB page, TLB misses are reduced, thus reducing this performance overhead. **Second**, in Linux, an address translation requires traversing 5-level page tables [16]. With the use of huge pages, TLB misses are less expensive, since address translations stop at the 4th level.

To demonstrate this effect empirically, we use Redis v4.0.11 [14], an in-memory key-value database, and the Linux perf tool [26], to monitor how THP affect a workload's execution. Specifically, we make 2 million *set* requests with 4KB objects in a fresh Redis server instance. We use Linux kernel 4.16.8 both with and without huge pages enabled and measure the number of **(1)** data and instruction TLB misses, **(2)** data and instruction cycles due to page walking caused by TLB misses, **(3)** main memory reads needed for these page walks and **(4)** total cycles and total execution time (Table 1).

| Counter | THP disabled | THP enabled |
|---|---|---|
| TLB data loads | 15,172,995,558 | 12,162,832,618 |
| TLB data load misses | 70,996,819 | 315,154 |
| TLB instruction load misses | 36,694,469 | 87,874 |
| TLB data store misses | 9,496,490 | 40,932 |
| Page walking data cycles | 1,358,301,181 | 18,422,086 |
| Page walking instruction cycles | 656,749,586 | 3,645,584 |
| DTLB page walker loads from memory | 227,534,040 | 421,743 |
| ITLB page walker loads from memory | 120,997,735 | 465,317 |
| Total cycles | 30,369,768,113 | 14,871,159,636 |
| Total execution time | 11.72s | 7.06s |

**Table 1: Profiling counters associated with TLB**

TLB misses cause page walks, a costly operation that reads processes' page tables, which involves reading data from multiple memory locations. In Table 1, we present the numbers associated with TLB misses, the memory loads and cycles due to page walking, and the total cycles and execution time. With THP disabled, TLB data load misses account for 0.47% of all data TLB loads, while with THP enabled, they account for only 0.0025% of total data TLB loads. Consequently, the reduction in TLB misses leads to fewer cycles and main memory accesses due to page walking, and results in a 1.66× speedup of the workload's execution.

## 2.4 Challenges

The use of huge pages is associated with a number of negative side effects which we present here.

**Increased page fault latency:** When a base page fault occurs, the Linux kernel's page fault handler tries to allocate a huge page instead. This increases the page fault latency for two reasons. **First**, the kernel has to find 2MB of contiguous physical memory to allocate and map a huge page. If the memory is highly fragmented during the request, the page fault handler tries to compact memory synchronously, to generate this amount of contiguous memory, which results in increased latency. The worst case scenario is when the page fault handler is not able to find 2MB of contiguous physical memory, even after the compaction is done, resulting in just allocating a base page with increased latency. **Second,** due to security reasons, the kernel has to zero out every page before giving them to processes, and a huge page needs more time to be zeroed.

We make 2 million *set* requests with 4KB objects in a fresh Redis-server instance. We run this experiment twice, once with the use of huge pages and once with base pages only, and present the results in Table 2. We use the ftrace tool [7] to trace the `__do_page_fault()` kernel function, to record the page fault latency in $\mu s$ and number of page faults.

| | Linux base pages only | Linux with huge pages |
|---|---|---|
| #page faults | 2,731,657 | 291,098 |
| Average | 0.888$\mu s$ | 2.851$\mu s$ |
| 90th | 1.501$\mu s$ | 1.744$\mu s$ |
| 99th | 2.805$\mu s$ | 118.232$\mu s$ |
| 99.9th | 4.201$\mu s$ | 123.78$\mu s$ |

**Table 2: Number and latency of page faults**

When huge pages are used, the number of page faults are decreased by approximately 90%, while the page fault latency increases compared with using base pages only. We observe that the 99th and 99.9th percentile of latency when huge pages are used are 42× and 29× the corresponding latency of using base pages only. This is a critical problem, since maintaining low tail latency is of utmost priority, particularly in data centers with hundreds to thousands of servers [17, 31], each with memory sizes that range from tens to hundreds of GBs. Thus, using `Linux` THP in its current form, is a double-edged sword. On the one hand, huge pages dramatically reduce the number of page faults incurred; on the other, it is unworkable for environments with vast memory.

**Memory Bloating:** The `Linux` kernel allocates a huge page on every base page fault. This aggressive policy causes memory bloating. Memory bloating occurs when a process reserves more memory than it uses, resulting in having an increased memory footprint. In our work, we define memory bloating as **(1)** having more allocated memory than the application actually requested and **(2)** keeping recently unutilized huge pages. To demonstrate this, in a system with 16GB of memory, we use Redis to make 2 million hset (hash set) requests with a 4KB object each, remove randomly 6GB, and then retrieve the remaining objects by triggering a *hgetall* (hash get all) command. When only base pages are used, this workload uses 7.6GB of memory; when huge pages are used, it consumes 11.1GB of memory. This bloat (46% more memory) distributes free pages all over physical memory, significantly reducing available contiguous memory. This directly impacts system performance because it increases huge page fault latency.

**Fragmentation:** The use of huge pages causes both internal and external fragmentation. Internal fragmentation is caused when a chunk of memory is not fully utilized. In the case of using huge pages, any process that needs only few pages, might end up reserving a greater amount of memory than it actually needs. External fragmentation occurs when free physical memory is divided in smalls chunks interleaved with allocated memory. Extensive memory fragmentation causes the most problems when transparent huge pages are used, since it leads to memory bloating and increased page fault latency.

**Huge pages are not swappable or migratable:** The current kernel implementation does not support huge-page swapping[2]. This means that a huge page has to be demoted to base pages, before swapping out these pages. This process degrades performance by delaying swapping and invalidating the corresponding TLB entry.

The `Linux` kernel's compaction algorithm migrates only base pages during compaction, while huge pages are not movable. This exacerbates the fragmentation problem, as it makes it difficult to maintain contiguous free memory.

## 2.5 Ingens

Ingens [15] is currently the state-of-the-art framework for huge page support on the `Linux` kernel. We briefly present how huge pages and memory compaction are handled in Ingens: **First**, if the number of base pages mapped in a huge page region is at least 90% of this region, then it promotes them to a huge page. This allows the promotion to occur asynchronously to avoid the costs of promoting during page faults. **Second**, Ingens provides fair sharing of huge pages using a metric based on the frequency of process accesses to both base and huge pages. Using this metric, it prioritizes promotions of processes that frequently access their allocated pages or that handle fewer huge pages. **Third**, Ingens periodically uses `Linux's` memory compaction algorithm to proactively compact memory to avoid fragmentation.

# 3 MEGA'S DESIGN

In this section, we provide a detailed description of the algorithms and mechanisms of our MEGA memory management framework. Our approach aims to maintain huge pages that are recently highly utilized and have most of their memory mapped. That is, we want to promote only huge pages that will offer a long-term performance gain; short-lived huge pages not only fail to provide benefits, but also hurt workload performance due to the cost of promotion and demotion. We refer to a 2MB block of memory as **huge page region** (HPR). Each huge page region in our system can be in one of the following states:

(1) **One or more base pages that are *mapped-tracked* but not *utilized-tracked*.** In this state, we just hold information about how many base pages are mapped in each huge page region. When the number of base pages mapped exceeds a given threshold, then we start tracking their utilization too and this huge page region proceeds to the next state.

[2] https://lwn.net/Articles/758677/

(2) **One or more base pages that are *mapped-tracked* and *utilized-tracked*.** In this state, we hold information about how many base pages are mapped and their utilization for a specific time interval. If the number of base pages mapped is below a specific threshold, then we stop tracking the utilization of this huge page region, falling back to the previous state. When the number of base pages mapped exceeds a given threshold and their accumulated utilization exceeds another threshold, then we promote this huge page region, which proceeds to the next state. We refer to a huge page region in this state as **huge page candidate**.

(3) **A huge page that is both *mapped-tracked* and *utilized-tracked*.** In this state, we hold information about the fraction of the memory that is allocated within this huge page and the huge page's utilization. When the amount of memory freed exceeds a given threshold or the utilization is below another threshold, then we demote this huge page, which falls back to the previous state.

## 3.1 Page mapping tracking

We now summarize the page mapping tracking process when a page fault occurs. To track which base pages are mapped in each huge page region, we use a bitmap of 512 bits, where each bit represents a base page in the huge page region. This bitmap is updated every time a base page is mapped by setting the corresponding bit of the huge page region to which it belongs. Similarly, when a base page is unmapped, we unset the corresponding bit. If a huge page region has at least **StartUTracking threshold** (50%, in our implementation) of its base pages mapped, we consider it to be a huge page candidate, and track its utilization, using the mechanism we describe in the next section. Similarly, we stop tracking its utilization if the percentage of base pages mapped drops below **StopUTracking threshold** (25%, in our implementation).

## 3.2 Huge page region utilization tracking

To track how much each huge page region is utilized, we use the Idle Page Tracking feature [36] introduced in `Linux` kernel 4.3. This feature allows us to track which memory pages were accessed in a certain amount of time via the Idle flag. We use the Idle flag in a three-step approach for the pages that we want to track: **1.** Set the flag. **2.** Wait for some time for these pages to be accessed. **3.** Check the flag.

In our implementation, we use a thread to periodically scan the utilization of huge page candidates and huge pages, to identify which of the former should be promoted and which of the latter should be demoted. After careful experimentation we chose this scan to occur every 5 seconds; our goal is to have a fresh snapshot of huge page candidates' utilization, while minimizing the cost of frequent TLB invalidations. Each huge page region that has its utilization tracked, is associated with a utilization history buffer, where we store the percentage of base pages that were accessed in each of the last N scans, where N=10 in our implementation. At the end of each scan, we average these percentages from the last N scans.

## 3.3 Promotion - demotion

Algorithm 1 presents the main reasoning behind our HPR utilization scan and promotion/demotion policies. Our promotion-demotion algorithm focuses on maintaining huge pages that are both heavily mapped *and* utilized. This is done by tracking which base pages are mapped in every huge page region, and periodically checking how much each huge page region is utilized. In particular, a huge page candidate is promoted only if its number of base pages mapped are over the **MProm threshold** (in our implementation, 90% of the base pages that a huge page region contains) and its accumulated utilization is over the **UProm threshold** (50% in our implementation) (Line 13-15). We chose these thresholds to reduce the drawbacks of using THP; by having over 90% of base pages mapped in a region, we restrict memory bloating, while requiring at least 50% utilization in the last 10 scans leads to the reduction of the overhead resulting from frequent promotions and demotions. Similarly, to avoid demotion, a huge page must have at least **MDem threshold** (50% in our implementation) base pages mapped and its accumulated utilization be at least **UDem threshold** (25% in our utilization). This means that if a huge page has fewer than 256 base pages mapped or its accumulated utilization drops below the **UDem threshold**, then we demote it (Lines 19-21). Promotion/demotion of huge page regions are costly, mainly due to TLB invalidations, and result in high CPU utilization. We select the aforementioned thresholds, after extended experimentation, to ensure that huge page regions are not frequently promoted/demoted. Our promotion/demotion mechanism allows us to asynchronously promote/demote huge page candidates, due to the fact that these operations are decoupled from the page fault process. This decoupling leads to improved page fault latency, since pages are faulted immediately.

## 3.4 Compaction

The problem with the approach `Linux's` kcompactd follows, is that it compacts memory when it is too late; even if there is plenty of free memory, it might be highly fragmented, preventing allocations. This problem is exacerbated when the kernel uses huge pages, because compaction causes increased page fault latency and negates the benefits of using

**Algorithm 1**
Periodic huge page region (HPR) utilization scan and promotion/demotion

```
1:  for all hpr ∈ util_tracked_hpr do
2:    hpr_util = 0
3:    if hpr is not huge_page then
4:      if #bpages_mapped < StopUTracking then
5:        Remove hpr from util_tracked_hpr
6:        return
7:      end if
8:      for all bpage ∈ hpr do
9:        if bpage not idle then
10:         hpr_util += 1
11:       end if
12:     end for
13:     if hpr_util ⩾ UProm and #bpages_mapped ⩾
        MProm then
14:       Promote hpr
15:     end if
16:   else
17:     if hpr not idle then
18:       hpr_util += 512
19:       if hpr_util < UDem or #bpages_mapped < MDem
          then
20:         Demote hpr
21:       end if
22:     end if
23:   end if
24: end for
```

huge pages. To illustrate why `Linux's` current memory compaction algorithm is inadequate, we use a workload to continuously allocate and free 10GB of memory in a machine with 16GB of memory. We run this experiment multiple times with different object sizes; we start with 8KB objects and for each different run we double the objects' size. In the last run, we use 128MB objects. We allocate 10GB to observe if `ZONE_NORMAL` (12GB of memory) can meet our requests. At the end of each run (after we have freed the allocated memory), we record the total available memory resulting from 2MB and 4MB blocks of contiguous available memory. We monitor blocks of these two sizes, because they can satisfy a request for a 2MB allocation, which is the size of a huge page. Additionally, we run concurrently a different workload that manipulates 420MB of sparsely allocated memory, to observe how it will affect the compaction. In Figure 1, we present the total combined available memory in MB as a combination of these blocks of contiguous available memory versus the object size.

We notice that from the first iteration, the available contiguous memory has dropped from approximately 10.9GB to

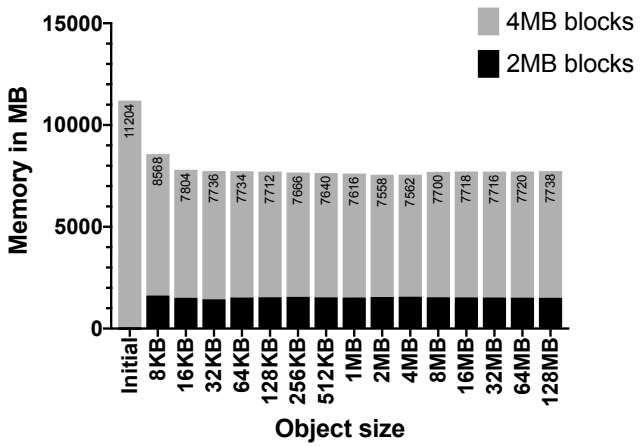


**Figure 1: Total contiguous available memory after a series of allocations/deallocations.**

approximately 8.3GB, despite the fact that we are performing multiple 8KB allocations and then free the allocated memory entirely. This amount of free available memory is fixed after every compaction; the memory cannot be completely compacted by the current compaction algorithm, despite the fact that every allocated object was freed. In the iteration where we allocate memory using 8MB objects, we notice that there is no sufficient contiguous memory to satisfy our requests; thus, some memory is allocated in `ZONE_DMA32`. Additionally, when ⩾ 32MB objects are used, there are no available 2MB or 4MB blocks inboth zones, requiring kcompactd to wake up to compact memory and make such blocks available. However, even after this compaction, only half of the requested huge pages could be allocated, due to memory fragmentation.

We propose a preemptive compaction algorithm that runs periodically (every 5s in our implementation) and each time tries to compact a small amount of memory (200MB in our implementation), to avoid the drawbacks of on-demand compaction. Our compaction algorithm aims both to proactively compact memory as well as to separate physical pages by their utilization and age of mapping. We seek to move, during compaction, pages that are less utilized, to avoid interfering with "hot" pages, which would lead to additional overhead. Additionally, we move older, in terms of mapping, pages based on the assumption that newly created data (memory) is more likely to "die" (be freed) in the near future.

Our compaction algorithm assumes that physical memory is divided into 2MB blocks, each one associated with a huge page frame number (HPFN), and for each one of them we keep track of **(1)** the time of the last base page mapping in the block, **(2)** the percentage of base pages mapped in the block in the interval between two iterations, and **(3)** the percentage of base pages accessed in the interval between

two iterations. Inspired by the cost-benefit approach used for segment cleaning in LFS [29], we calculate at the end of each pass a cost-benefit ratio for every block, as follows:

$$\frac{benefit}{cost} = \frac{age * (1 - \%bpagesMapped) * (1 - \%bpagesAccessed)}{(2 * \%bpagesMapped)}$$

where *age* is the time of the last base page mapped, *1-bpagesMapped* is the percentage of base pages not mapped in the current scan, *1-bpagesAccessed* is the percentage of base pages not accessed in the current scan, and *2*bpagesMapped is the cost of migrating this block (bpagesMapped to read and bpagesMapped to write).* Because memory allocations happen primarily in ZONE_NORMAL, we perform this process for ZONE_NORMAL on every compaction run. ZONE_DMA32 is used mostly by hardware that can only do DMA using 32-bit addresses. For this zone, we only compact memory if the number of contiguous blocks that can satisfy a 2MB allocation is less than 33% of this zone's available free memory. Our rationale behind this is to minimize the interference and memory compaction cost as much as possible.

# 4 EVALUATION

In this section, we evaluate MEGA by comparing it with the `Linux` 4.16.8 kernel and Ingens, which is the state-of-the-art in huge page management, implemented in `Linux` kernel 4.3. We use the Ingens' code that is available on github [5]. We run most of our experiments on a system (Intel i7 @ 2,3GHz, 16GB DDR3 RAM, 500GB SSD), unless explicitly stated. Our work consists of 1.4 KLoC and is available in [38]. KPTI [37] is enabled on the default `Linux` kernel 4.16.8 and MEGA.

Our goal is to answer the following questions:

- What effect on page latency does MEGA have compared with `Linux` THP and Ingens?
- How do frequent promotions/demotions in close temporal proximity affect workload performance, and why should we avoid them?
- In which cases does MEGA underperform compared with Ingens?
- How efficiently does MEGA's memory compaction algorithm tackle memory fragmentation and preserve contiguity?

## 4.1 Page fault latency

We repeat the experiment of Section 2.4 to measure page fault latency with the use of the ftrace tool. We compare **(1)** `Linux` 4.16.8 with base pages only, **(2)** `Linux` 4.16.8 with huge pages enabled, **(3)** Ingens and **(4)** our MEGA implementation. Table 3 shows the average page fault latency incurred for each system as well as the 90th, 99th, and 99.9th latency percentiles. The numbers in parentheses indicate the slowdown (i.e., the factor by which page fault latency increases) with respect to `Linux` when it uses base pages only. We observe that MEGA incurs a 2.06-3.59X slowdown when compared with `Linux` using base pages only. This is to be expected given the extra bookkeeping MEGA must perform to track page mappings, utilization, and information required for memory compaction. On the other hand, `Linux` with huge pages sees a dramatic performance degradation with a 1.2-42.21X slowdown. Similarly, while Ingens has the lowest slowdown for average, 90th and 99th percentiles for latency, its 99.9th percentile latency is drastically higher, suffering a 95X slowdown. Given the importance of low tail latencies [17, 31], we believe MEGA is preferable as it keeps slowdown to a moderate level, even for its tail latencies (0.05-0.12X when compared with `Linux` using huge pages). This is the result of decoupling huge page promotion from the page fault handling procedure.

| | Linux base pages only | Linux Huge Pages | Ingens | MEGA |
|---|---|---|---|---|
| Average | 0.9 | 2.9 (x3.22) | 1.6 (x1.78) | 2.5 (x2.78) |
| 90th | 1.5 | 1.8 (x1.2) | 1.7 (x1.13) | 3.1 (x2.06) |
| 99th | 2.8 | 118.2 (x42.21) | 4.5 (x1.60) | 6.1 (x2.17) |
| 99.9th | 4.2 | 123.8 (x29.46) | 400.8 (x95.42) | 15.1 (x3.59) |

**Table 3: Page fault latency in $\mu$s. In the parentheses, the slowdown is shown with respect to `Linux` when it uses base pages only.**

## 4.2 Utilization based promotion - demotion

To demonstrate the efficiency of MEGA's utilization-based promotion and demotion approach, we run two different workloads on MEGA and on Ingens. In the first workload, we allocate 6 GB of memory in chunks of 32MB and use only one sixth of it (1GB); we run two instances of this workload concurrently to put pressure on the system. In Table 4 we record **(1)** how many huge pages are used by the workloads during the execution, and **(2)** how many 2MB blocks are available before and after one minute of execution.

| Metric | Ingens | MEGA |
|---|---|---|
| #THP used | 2,447 | 1,024 |
| #2MB blocks before execution | 6,710 | 6,733 |
| #2MB blocks after 1 min. of execution | 50 | 282 |

**Table 4: Number of THP used and number of contiguous free 2MB blocks**

We notice the following. **First**, Ingens uses 2447 huge pages, while MEGA needs less than half (1024) the number of huge pages to fully exploit the benefits of their use. Ingens' high number of huge pages puts pressure on the system, for two reasons: **(1)** It forces the system to find and allocate multiple 2MB blocks to satisfy the requests, and **(2)** It reduces the number of available 2MB blocks, preventing other workloads

from potentially utilizing them more effectively. **Second**, according to the Ingens promotion policy, the number of huge pages used should be approximately 6144 (2 workloads * 6GB allocated/2MB per huge page). This significant deviation is the result of two things. First, the memory is extremely fragmented, thus, it is not possible to allocate additional huge pages. Second, when it is time for Ingens to promote a huge page candidate, it makes the appropriate kernel calls and a flag is set in Ingens code, denoting that this region is a huge page, to avoid false future calls for promotion. The same code path happens for demotion. However, there is no check over whether the promotion/demotion failed, resulting in incorrect handling of pages. On the other hand, in our case, the workloads allocate only base pages. When the utilization is over the threshold, MEGA tries to allocate 1024 huge pages, however due to fragmentation, it allocates only a fraction of them. After some time, the kernel retries and succeeds in allocating the rest of the needed huge pages. Note that Ingens manages to allocate a large number of huge pages from the start, due to the fact that it requests them before the whole amount of memory needed by the workloads is allocated. **Finally**, although not shown in Table 4 due to lack of space, in Ingens, the number of 2MB blocks continues to be only 50 (100MB) even after several minutes beyond the end of the 1 minute experimentation period; this is because memory has become extremely fragmented. In MEGA, the number of 2MB blocks continuously increases. We evaluate this behavior further in Section 4.3.

Next, we evaluate the performance overhead of frequent promotions/demotions. We allocate 8GB of memory, iterate over it briefly and then free the whole allocated memory. We do this 50 times in Ingens and MEGA and report total execution time in Table 5. Due to its continuous promotions and demotions, Ingens has an execution time that is 1.58× greater than MEGA's execution time. In contrast, MEGA's utilization-based approach benefits workloads that use chunks of memory even for a short amount of time.

| | **Total execution time in seconds** |
|---|---|
| Ingens | 47.6s |
| MEGA | 30s |

**Table 5: Overhead due to frequent promotions - demotions**

Finally, we demonstrate an extreme case, where MEGA displays suboptimal performance compared with Ingens. We allocate 6GB and visit the data in steps of 32 * 1024, i.e. the data in memory location 0, 32 * 1024, 64 * 1024 etc, then the data in memory location 8, 32 * 1024 + 8, 64 * 1024 + 8 etc. We choose this step deliberately because it is equal to the L1 data cache size. We run this workload 10 times and display the total execution time in Table 6.

| | **Total execution time in seconds** |
|---|---|
| Ingens | 158.8s |
| MEGA | 324.1s |

**Table 6: Extreme case of not using THP in MEGA**

MEGA's execution time is approximately twice Ingens' execution time due to the fact that the former does not use huge pages. This is expected because in MEGA, this access pattern does not allow any huge page region to exceed the utilization threshold. However, we consider these erratic patterns to be improbable, and we expect that most workloads will allow huge page regions to build up considerable utilization levels, allowing them to be promoted to huge pages.

## 4.3 Compaction

As mentioned earlier, fragmentation is the major cause of overhead when huge pages are used. Ingens uses proactive compaction to maintain free contiguous memory at all times and reduce fragmentation as much as possible. Specifically, it does a periodic scanning every 5 seconds and tries to compact up to 100MB per zone (for `ZONE_DMA32` and `ZONE_NORMAL`). We compare the Ingens compaction algorithm with MEGA's compaction algorithm using the default parameters of running every 5 seconds and trying to compact up to 200MB. We allocate 12GB of memory for our workload, while the kernel uses 1.5GB of memory, leaving 2.5GB available for use. We iterate through the data, then free 50% of the allocated memory in chunks of 1MB and continue iterating the remaining memory. We execute this experiment for 3 minutes and record how many 2MB blocks are available. In the first two minutes we observe how the compaction algorithms behave under memory pressure, while in the last minute we stop the program's execution and record how fast the system restores 2MB blocks. In both cases, the system initially has approximately 6800 2MB blocks available. Additionally, we record the values of the major compaction parameters the kernel tracks (mentioned in Section 2.2) to monitor how well the algorithms perform internally.

Figure 2 shows the number of available 2MB blocks in memory as a function of time for Ingens and MEGA. We observe that Ingens' compaction algorithm is not able to restore 2MB contiguous blocks as fast as they are needed, neither under pressure nor when nearly all available memory is free. Additionally, there is a small decline of available blocks 40 seconds after the start of the experiment. This behavior can be explained by the fact that Ingens prioritizes each time the blocks that are closer to the start of the address space, regardless of their utilization. On the contrary, in both cases, MEGA correctly identifies blocks that are less frequently utilized and/or old and compacts them, nearly doubling the available 2MB contiguous blocks throughout the duration of the experiment.

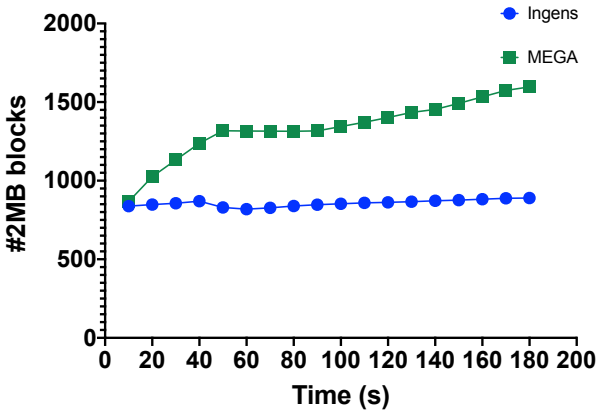


**Figure 2: Total contiguous free blocks 2MB blocks during the experiment**

| Statistics | Ingens per scan | MEGA per scan |
|---|---|---|
| Pages scanned for migration | 55,256 | 7,537 |
| Free pages scanned | 149,993 | 350,702 |
| Isolated pages | 2,918 | 14,906 |
| Successfully migrated pages | 1,432 | 7,352 |
| Failed migrated pages | 1.16 | 0.94 |

**Table 7: Memory compaction statistics**

Table 7 shows the average compaction's statistics per scan. The lower number of pages scanned for migration and higher number of free pages scanned is expected, since our MEGA algorithm scans only specific blocks of memory for compaction, restricting the page migration scanner and expanding the chunk of memory the free page scanner searches. In conclusion, we observe that for the same targeted amount of memory, MEGA achieves approximately 5x the number of successfully migrated pages and 0.8x number of failed migrated pages, compared to Ingens.

## 4.4 Compute-Intensive Workloads

We use the PARSEC 3.0 benchmark suite to measure the performance impact that MEGA and Ingens have on typical compute intensive workloads that do not benefit from the use of huge pages. Figure 3 shows the normalized execution time of MEGA and Ingens compared with `Linux` with huge pages enabled. In spite of the extra bookkeeping effort that MEGA entails, we observe that MEGA achieves in all, but one, workloads an execution time speedup of up to 9.5% over `Linux`, while Ingens mostly displays an additional overhead.

We also measure the performance and latency of MySQL [18] using the sysbench benchmark suite [32]. We run for 1 minute a read-only workload on a table with 30 million rows, executed by 8 threads, and record min, average, max

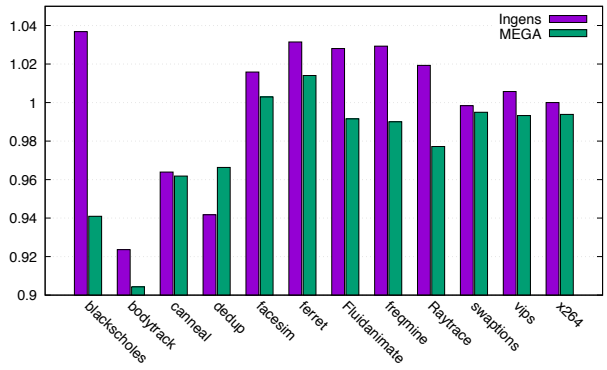


**Figure 3: Execution time of MEGA and Ingens normalized w.r.t Linux (lower is better)**

and 95th latency, and the number of transactions per second for MEGA, Ingens, and `Linux` kernel 4.16.8 both with base pages only and with huge pages.

| | MEGA | Ingens | Linux base | Linux huge |
|---|---|---|---|---|
| Min | 0.76$\mu$s | 0.68$\mu$s | 0.74$\mu$s | 0.85$\mu$s |
| Average | 1.44$\mu$s | 1.71$\mu$s | 1.63$\mu$s | 1.32$\mu$s |
| Max | 54.01$\mu$s | 108.12$\mu$s | 11.66$\mu$s | 64.94$\mu$s |
| 95th percentile | 2.18$\mu$s | 3.62$\mu$s | 3.07$\mu$s | 1.76$\mu$s |
| Transactions per sec | 5556.29 | 4661.36 | 4895.21 | 6056.84 |

**Table 8: Sysbench results for MEGA, Ingens and the default `Linux` kernel with/without THP enabled**

We observe that Ingens experiences the biggest average, max and 95th latency and the lowest transaction throughput (in transactions per second). On the contrary, MEGA combines the benefits of both using huge pages and base pages only; the number of transactions per second is close to the number of transactions per second that `Linux` with huge pages achieves, while keeping the latency at low levels.

## 4.5 Big-Memory Workloads

In this experiment, we evaluate how MEGA behaves with big-memory workloads on a much larger machine (Intel Xeon Processor E5-2650 v4 @2,2GHz, 256GB DDR4 RAM, 2.8TB of SATA 3 SSD). Initially, we run, on a freshly booted machine, a simple workload that allocates 200GB, iterates briefly over the allocated memory and then frees half of it (100GB) in chunks of 1MB. We then run 5 iterations where we allocate 100GB and then free the memory allocated in the previous iteration (i.e., the first iteration frees the initial 100GB that remained, the second iteration frees the 100GB allocated in the first iteration, etc). This process simulates cloud systems whose memories quickly become fragmented as a result of running real client workloads [1]. After the last

iteration, the memory has 100 GB allocated, which leaves approximately 156GB of sparsely available memory. We then run, in a fresh Redis server instance, the redis-benchmark [3] with 40000 set operations with 12-byte sized keys that are chosen randomly from a 7536640-sized key space, values of 2MB and 50 parallel clients. The purpose of this experiment is to test under a real-life scenario: **(1)** how Redis throughput and latency are affected, and **(2)** what happens to the 2MB and 4MB blocks. Note, we choose this value size to investigate how easily the system can find 2MB of contiguous available memory, while we deliberately create "holes" that are half the size of the payload. We run this experiment on both MEGA and Ingens and show the results in Table 9. We observe that Redis over Ingens experiences a 27% slowdown in execution time compared with Redis over MEGA . This is because Ingens more aggressively promotes/demotes huge pages in the first workload and extensively fragments the available memory, while its compaction algorithm is unable to restore the desired amount of contiguous available memory in time.

| Stats | Ingens | MEGA |
|---|---|---|
| Throughput (req/s) | 501.54 | 638.05 |
| 99th latency | 278ms | 104ms |
| 99.9th latency | 421ms | 260ms |
| 99.99th latency | 505ms | 266ms |
| Execution Time | 79.75s | 62.69s |

**Table 9: Redis over Ingens and MEGA under a real-life scenario**

Moreover, Table 10, shows the number of available 2MB and 4MB blocks. We continue to monitor these numbers for 5 minutes after the redis-benchmark has ended, and we confirm again that Ingens cannot keep up with MEGA's block restoration rates; Ingens' numbers of 2MB and 4MB blocks remain the same, while MEGA continues to increase these numbers. Finally, even though there was more than enough available memory, both for Redis' internal structures and the key-values to be stored, increasing the number of set operations in Ingens causes the first workload to be killed due to extreme memory starvation, while this does not happen in MEGA.

| Memory block size | Ingens | MEGA |
|---|---|---|
| #2MB | 42 | 14 |
| #4MB | 58 | 1146 |
| Total available 2MB blocks | 158 | 2306 |

**Table 10: Number of available 2MB/4MB blocks at the end of the experiment**

## RELATED WORK

Navarro et al. [19] introduced superpage management in FreeBSD, using reservation-based allocations and contiguity-aware page replacements. Panwar et al. [25] focus on memory fragmentation control, show that unmovable pages are a major fragmentation issue, and introduce a memory manager that provides an unmovable page tracker and a modified page allocator. Ingens [15] is the state-of-the-art framework for huge page support in `Linux` and has served as the inspiration for our work. From Ingens, MEGA borrows the use of percentage of based pages mapped as a criterion for promotion, however, MEGA also takes into account the utilization of huge page regions. We have found that using both of these criteria, in combination, is effective, and thus, an important missing element from modern memory management. Moreover, for memory compaction, Ingens simply uses the default `Linux` kernel mechanism, i.e., it periodically compacts the 100MB lowest megabytes of memory. In contrast, MEGA uses a completely different algorithm that tracks information about physical memory and uses a cost-benefit approach to decide which physical 2MB regions to compact. This new algorithm allows MEGA to alleviate the memory fragmentation problem, while minimizing the cost of compaction and the interference amongst workloads that use the compacted regions.

Overall, prior techniques aim to solve only a subset of the problems of huge page use. Our work aims to provide a general solution by combining monitoring of memory and effective fragmentation handling through a novel compaction algorithm to tackle these problems. Additionally, to the best of our knowledge, we are the first to examine and tackle the problem of huge page promotions and demotions in close temporal contiguity.

## CONCLUSION

Effective huge page management is crucial as applications become exponentially more demanding in their memory needs over time. We believe MEGA is an important step in this direction. Our plans for future work include extending, with the MEGA framework, other major operating systems that have primitive huge page management. Additionally, we will extend our memory compaction algorithm, by making it aware of non-movable areas of memory. Specifically, our goal is to distinguish movable and non movable pages, to reduce the compaction cost and improve the fragmentation problem.

## ACKNOWLEDGMENTS

We would like to thank our paper shepherd, Michael Swift and the anonymous reviewers for their insightful feedback.

## REFERENCES


[1] Jean Araujo, Rubens Matos, Paulo Maciel, Rivalino Matias, and Ibrahim Beicker. 2011. Experimental Evaluation of Software Aging Effects on the Eucalyptus Cloud Computing Infrastructure. In *Proceedings of the Middleware 2011 Industry Track Workshop (Middleware '11)*. ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/2090181.2090185

[2] 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)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[3] Redis benchmark utility. Accessed 2019. https://redis.io/topics/benchmarks

[4] Abhishek Bhattacharjee. 2013. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 383–394. https://doi.org/10.1145/2540708.2540741

[5] Coordinated and Efficient Huge Page Management system. Accessed 2019. https://github.com/ut-osa/ingens

[6] A detailed description about physical memory's zones. Accessed 2019. https://notes.shichao.io/lkd/ch12/#zones

[7] ftrace. Accessed 2019. https://elinux.org/Ftrace

[8] Windows' VirtualAlloc function. Accessed 2019. https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualalloc

[9] Mel Gorman and Andy Whitcroft. 2006. The what, the why and the where to of anti-fragmentation. (01 2006).

[10] Transparent huge pages in 2.6.38. Accessed 2019. https://lwn.net/Articles/423584

[11] Oracle: Disabling Transparent HugePages. Accessed2019. https://docs.oracle.com/en/database/oracle/oracle-database/12.2/ladbi/disabling-transparent-hugepages.html#GUID-02E9147D-D565-4AF8-B12A-8E6E9F74BEEA

[12] An introduction to Linux memory compaction. Accessed 2019. https://lwn.net/Articles/368869

[13] 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)*. ACM, New York, NY, USA, 66–78. https://doi.org/10.1145/2749469.2749471

[14] Redis key-value store. Accessed 2019. http://redis.io

[15] 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)*. USENIX Association, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[16] Five level page tables. Accessed 2019. https://lwn.net/Articles/717293

[17] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2670979.2670988

[18] MySQL. Accessed 2019. https://en.wikipedia.org/wiki/MySQL

[19] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation (OSDI '02)*. USENIX Association, Berkeley, CA, USA, 89–104. http://dl.acm.org/citation.cfm?id=1060289.1060299

[20] Summary of hugetlbpage support in the Linux kernel. Accessed 2019. https://docs.mongodb.com/manual/tutorial/transparent-huge-pages

[21] Man page for Linux mmap. Accessed 2019. http://man7.org/linux/man-pages/man2/mmap.2.html

[22] Man page for Linux shmget. Accessed 2019. http://man7.org/linux/man-pages/man2/shmget.2.html

[23] Man page for OSX mmap. Accessed 2019. https://www.unix.com/man-page/osx/2/mmap

[24] Redhat: Huge Pages and Transparent Huge Pages. Accessed 2019. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge

[25] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[26] perf. Accessed 2019. https://en.wikipedia.org/wiki/Perf_(Linux)

[27] Page Frame Reclamation. Accessed 2019. https://www.kernel.org/doc/gorman/html/understand/understand013.html

[28] Linux release 2.6.38. Accessed 2019. https://kernelnewbies.org/Linux_2_6_38

[29] Mendel Rosenblum and John K. Ousterhout. 1991. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/121132.121137

[30] Windows' Large-Page Support. Accessed 2019. https://docs.microsoft.com/en-us/windows/desktop/memory/large-page-support

[31] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 513–527. http://dl.acm.org/citation.cfm?id=2789770.2789806

[32] sysbench benchmark suite. Accessed 2019. https://github.com/akopytov/sysbench

[33] Cloudera: Disabling Transparent Hugepages (THP). Accessed 2019. https://www.cloudera.com/documentation/enterprise/5-9-x/topics/cdh_admin_performance.html#cdh_performance__section_hw3_sdf_jq

[34] Couchbase: Disabling Transparent Huge Pages (THP). Accessed 2019. https://docs.couchbase.com/server/6.0/install/thp-disable.html

[35] MongoDB: Disable Transparent Huge Pages (THP). Accessed 2019. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

[36] Idle Page Tracking. Accessed 2019. https://www.kernel.org/doc/html/latest/admin-guide/mm/idle_page_tracking.html

[37] Wikipedia. 2019. https://en.wikipedia.org/wiki/Kernel_page-table_isolation

[38] MEGA: Overcoming Traditional Problems with OS Huge Page Management. Accessed 2019. https://github.com/Tmichailidis/MEGA