Making Huge Pages *Actually* Useful

Ashish Panwar[*]
Indian Institute of Science
panwarit014@gmail.com

Aravinda Prasad[†]
Indian Institute of Science
aravinda@iisc.ac.in

K. Gopinath
Indian Institute of Science
gopi@iisc.ac.in

Abstract

The virtual-to-physical address translation overhead, a major performance bottleneck for modern workloads, can be effectively alleviated with huge pages. However, since huge pages must be mapped contiguously, OSs have not been able to use them well because of the memory fragmentation problem despite hardware support for huge pages being available for nearly two decades.

This paper presents a comprehensive study of the interaction of fragmentation with huge pages in the Linux kernel. We observe that when huge pages are used, problems such as high CPU utilization and latency spikes occur because of unnecessary work (e.g., useless page migration) performed by memory management related subsystems due to the poor handling of unmovable (i.e., kernel) pages. This behavior is even more harmful in virtualized systems where unnecessary work may be performed in both guest and host OSs.

We present Illuminator, an efficient memory manager that provides various subsystems, such as the page allocator, the ability to track all unmovable pages. It allows subsystems to make informed decisions and eliminate unnecessary work which in turn leads to cost-effective huge page allocations. Illuminator reduces the cost of compaction (up to 99%), improves application performance (up to 2.3×) and reduces the maximum latency of MySQL database server (by 30×).

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

Keywords Address Translation; Memory Fragmentation; Memory Compaction; Huge Pages

ACM Reference Format:
Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages *Actually* Useful. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3173203

1 Introduction

Address translation overhead has become a major performance bottleneck for modern data processing applications as TLB scaling has not caught up with the growth of memory capacity in recent times [31, 40, 49, 63]. This has inspired support for huge pages in most processors [4, 5, 53]. Huge pages minimize CPU time spent in performing page table lookups by reducing TLB misses [29, 46].

Unfortunately, the profitability of huge pages depends on the state of memory fragmentation as they must be mapped in contiguous memory [49, 54]. In long-running systems, unfavorable fragmentation can (and often does) become a source of poor performance.

Users have repeatedly experienced performance degradation, high kernel space CPU utilization and latency spikes while using huge pages with important applications such as Hadoop, MongoDB, Redis and VoltDB [16, 17, 19, 20]. To avoid such issues, most database servers are shipped with huge pages disabled. Hence, we can safely infer that the huge page support available in hardware for nearly two decades is not effectively utilized by OSs. Contrary to a popular belief in the academic literature that fragmentation is not a critical problem and OSs can efficiently recover from fragmentation with memory compaction (i.e., by relocating pages) [29, 31, 46], we show that fragmentation can indeed cause the aforementioned issues with huge pages. Our views are also well corroborated by the Linux kernel community discussions [6, 7].

This paper presents a comprehensive study of the interaction of fragmentation with huge pages in the Linux kernel. We find that the poor handling of unmovable (i.e., kernel) pages, which are found in many OS designs, is the root cause of many huge page related problems. While previous work indicates that the existing OS designs can effectively handle unmovable pages [59], we believe this to be erroneous at least in the context of long-running systems.

We identify two major issues to be the root cause of various performance anomalies that appear with huge pages: 1) *fragmentation via pollution*, which occurs when memory contiguity is unnecessarily polluted with unmovable pages,

[*] Currently at NetApp, ashish.panwar@netapp.com
[†] Currently at IBM.

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.
ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-4911-6/18/03...$15.00
https://doi.org/10.1145/3173162.3173203

and 2) *latency-inducing unsuccessful (LIU) migration*, which occurs when the kernel unnecessarily migrates pages from the polluted regions while attempting to allocate huge pages. While the former often leads to severe fragmentation, the latter induces latency by increasing the cost of recovering from fragmentation. As fragmentation via pollution increases, the overhead of LIU migration starts to dominate the benefits of huge pages. Importantly, both fragmentation via pollution and LIU migration occur due to poor decision making in memory management related subsystems.

With modest changes in the Linux kernel, Illuminator explicitly tracks all unmovable pages to help various subsystems in avoiding unnecessary work. For example, it helps memory compaction in avoiding LIU migration which leads to cost-effective huge page allocations. It also helps the page allocator in efficiently clustering unmovable pages which in turn reduces fragmentation via pollution. These optimizations provide significant performance improvement across native and virtualized systems.

This paper makes the following major contributions:

- We identify how the existing memory management policies lead to various performance anomalies often experienced in the real world with huge pages (§ 4).
- We present Illuminator, a simple memory management framework that effectively mitigates fragmentation and makes huge page benefits accessible to applications even in stressful conditions (§ 5).
- Through a detailed evaluation, we show that Illuminator significantly outperforms Linux in terms of various performance metrics such as the execution speedup, latency, jitter and performance isolation (§ 6).

2 Motivation

Huge pages have become vital for mitigating address translation overheads of modern workloads as non-linear scaling of the hardware cost, energy consumption and lookup latency has impacted the growth of TLB capacity [40, 49, 63]. Though each address translation may require up to 6× more memory lookups in virtualized systems due to the two-dimensional page tables [31], modern hardware can nearly match the address translation efficiency of virtualized environments with the native systems by utilizing huge pages [61]. However, harnessing the full potential of huge pages requires efficient fragmentation mitigation by the OSs.

OSs can leverage the hardware support for huge pages in two ways. First, by reserving a pool of contiguous memory at boot time. Windows and OS X support huge pages using this mechanism. Linux provides a similar `libhugetlbfs` interface for using huge pages in well-tuned system [12]. However, this approach is not flexible and necessitates application modifications to explicitly request huge pages [63]. In practice it is tedious for developers to know which memory regions should (or should not) be backed by huge pages.

Second, a more practical and widely accepted approach is Transparent Huge Pages (THP), where an OS tries to allocate huge pages automatically as in Linux and FreeBSD. Unfortunately, the profitability of THP support declines as a system runs longer. We discuss some examples to highlight a few issues users often face with huge pages.

Tales from the real world: Customers with large Helix user bases experienced unresponsive servers and frozen processes despite being provisioned with adequate memory [23]. Troubleshooting revealed that THP feature of the Linux kernel was the culprit because `khugepaged` process (§ 3.2) was consuming a high number of CPU cycles. Even simple commands (e.g., *ps*) appeared to be frozen [9]. Disabling huge pages solved the problem.

In many cases the first recommendation for avoiding many-msec hiccups is to disable THP support in Linux as the kernel thread interferes with latency-sensitive applications [19, 24]; we have also observed as much as 4.7 seconds hiccups (see Figure 10). The Hadoop Performance Tuning Guide recommends disabling huge pages because compaction increases the kernel space CPU utilization by 66% for the TeraSort benchmark [16]. Many popular database vendors also recommend disabling huge pages, primarily to avoid latency spikes [17–20, 25].

We show that these issues can appear due to the poor handling of unmovable pages which can be found in many OSs such as FreeBSD, Solaris and OS X [2, 47, 52, 54]. Importantly, several OSs do not support THP primarily to avoid dealing with fragmentation [10]. Illuminator makes THP support actually useful for long-running systems. While we believe the Illuminator design to be useful for different OSs, we present it here in the context of the Linux kernel.

3 Memory management background

3.1 Unmovable pages

The mobility of memory pages depends on the reference management structures. The user virtual address space is typically managed with page tables and a few other objects (e.g., *vm_mm, vm_area_struct* etc. [33]). These pages (including non-pageable *mlocked* regions) can be migrated by copying their content and updating the corresponding references. In contrast, the kernel address space is directly mapped into physical memory which provides faster mapping of kernel objects: a kernel virtual address is converted to a physical address (or vice-versa) by adding a constant. The direct mapping leads to a simple design whose benefits are well documented in the literature [8, 11, 29]. However, it makes the kernel objects (e.g., inodes) unmovable—directly mapped objects cannot be migrated or swapped to disk as their references cannot be tracked. While unmovable pages are not unique to Linux,[1] the severity of fragmentation caused by

[1]The outermost page tables and DMA pages are inherently unmovable in most OS designs.

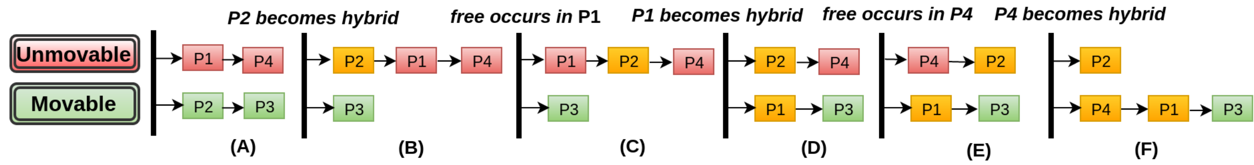

Figure 1. Two-way classification based anti-fragmentation leads to fragmentation via pollution because the buddy allocator cannot reuse hybrid pageblocks during fallbacks. For example, pollution of P1 can be avoided by reusing P2 during C⟶D. For clarity, hybrid pageblocks are colored *yellow* in this figure, but the Linux kernel treats them as either *red* or *green*, depending on which region they belong to.

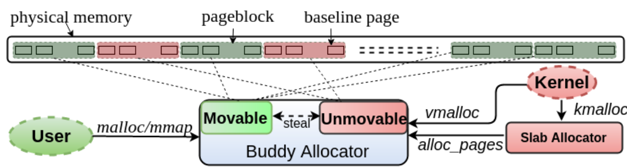

Figure 2. Physical memory management in Linux. The slab allocator allocates kernel objects while the buddy allocator serves pages to the slab allocator (from the unmovable region) and to the user space (from the movable region).

unmovable pages is high in direct mapped kernels as all kernel pages are unmovable in such systems.

3.2 Memory allocation

Figure 2 shows the basic memory allocation framework of Linux. Kernel objects are allocated by the slab allocator which maintains separate caches (known as slab caches) for different objects. Slab caches are populated by requesting memory from the buddy allocator via `alloc_pages`. Note that *pages occupied by the slab caches are unmovable.*

Application requests are handled by the buddy allocator which preferentially allocates huge pages at the time of page fault, using direct (synchronous) compaction if memory is fragmented. If huge page allocations fail, the kernel allocates baseline pages which may be promoted to huge pages in the background by the kernel thread khugepaged using asynchronous compaction. We use the term page(s) and baseline page(s) interchangeably.

3.3 RCU and deferred objects

Many subsystems using the slab allocator employ Read-Copy-Update (RCU) synchronization mechanism for better scalability. In RCU, writers create a new copy before updating an object and defer the freeing of the old copy. The deferred object is reclaimed by the synchronization mechanism after a *grace period*, which denotes the completion of all its pre-existing readers. Notice that delay in reclaiming deferred objects can increase unmovable memory footprint as these objects are unmovable.

3.4 Fragmentation mitigation techniques

Unfavorable placement of unmovable pages can create permanent fragmentation leading to huge page allocation failures. Current systems employ a combination of fragmentation avoidance (i.e., anti-fragmentation [55]) and recovery mechanism (i.e., memory compaction) to handle fragmentation. We briefly discuss these mechanisms below.

Anti-fragmentation clusters alike allocations by dividing memory in two disjoint regions (see Figure 1): *unmovable* (colored red) and *movable* (colored green). This partitioning is done at pageblock granularity; a pageblock represents a huge page sized contiguous physical memory region. The buddy allocator selects a region based on the source of the allocation request. For example, kernel requests are served from the unmovable region. However, a region can steal pageblocks from the other region when it runs out of free memory (this event is referred to as fallback). During fallback, the stolen pageblock is colored accordingly. For instance, the color of a pageblock is updated to red if it is stolen by the unmovable region.[2]

The memory compaction algorithm involves two pointers; from one end, the *migrate scanner* prepares a list of in-use and movable baseline pages while the *freepage scanner* collects free baseline pages from the other end [6]. Pages from the migrate scanner's list are copied to the freepage scanner's list to restore memory contiguity.

4 A detailed analysis of fragmentation

The virtual memory layer in the Linux kernel has undergone significant changes for accommodating the support for huge pages. However, operations at the physical memory layer (i.e., page/object allocation, compaction) have largely remained unchanged. This section discusses how unnecessary work can occur and become a source of poor performance. First, we introduce the notion of a hybrid pageblock.
Hybrid pageblock: A pageblock is hybrid if it contains both movable and unmovable pages. We explain how hybrid pageblocks are formed with an example (see Figure 1).

[2]Real implementation of anti-fragmentation has more than two regions in the Linux kernel [54, 55]. However, to simplify the discussion in this paper, we describe it only in the context of movable and unmovable regions.

Let us assume a system starts with four pageblocks P1 to P4. The two-way classification approach works well until the system reaches state A where P1, P4 belong to the unmovable region and P2, P3 belong to the movable region. If P1 and P4 have no free pages left and an unmovable page is requested by the kernel, the algorithm of Linux steals and adds P2 to the unmovable region during system transition from A→B. P2 thus becomes a hybrid pageblock during A→B, if some movable pages were already allocated from it.

4.1 The invisibility of hybrid pageblocks

Two-way classification based anti-fragmentation makes hybrid pageblocks invisible which in turn represents a *stale* view of movability. While there are three types of pageblocks (i.e., movable, unmovable and hybrid) in the system, subsystems operate on the assumption of only movable and unmovable pageblocks. This leads to the following scenarios of poor decision making in critical code paths.

4.1.1 Fragmentation via pollution

This is a situation where unmovable pages are *unnecessarily* allocated from movable (green) pageblocks. This happens because even though hybrid pageblocks are placed at the head of free lists during fallbacks, they get shifted away from the head over a period of allocation and free cycles. Thus, the buddy allocator cannot identify or reuse existing hybrid pageblocks efficiently. For example in Figure 1, P1 gets polluted during C→D because the buddy allocator is not aware of P2 already being a hybrid pageblock. Similarly, the transition from E→F also produces a new hybrid pageblock P4. In the worst case, each fallback can produce a fresh hybrid pageblock in the system.

Fragmentation via pollution restricts huge page allocations since a polluted pageblock cannot be allocated as a huge page unless its unmovable pages are freed. In practice, only a few misplaced unmovable pages can create permanent fragmentation. For example, a 2MB pageblock on x86 systems consists of 512 baseline pages (512×4KB=2MB). However, a single unmovable page is sufficient to pollute a pageblock and hence only 0.19% misplaced unmovable pages (of total system pages) can pollute all the pageblocks (1/512=0.0019) in the worst case. Moreover, the severity of fragmentation via pollution increases along with the size of huge pages — a system using 1GB-sized huge pages can be fragmented by only 0.00038% misplaced unmovable pages. In this paper we discuss only the most commonly used 2MB huge pages for simplicity.

4.1.2 LIU migration

During compaction, the kernel migrates pages from all green pageblocks encountered by the migrate scanner, optimistically believing that a green pageblock can always be emptied. But many hybrid pageblocks are also colored green in the two-way classification of pageblocks (for example, both P1 and P4 are hybrid in state F). This behavior leads to LIU migration.

# Unmovable Pages	# Hybrid Pageblocks	
	Linux (664)	Illuminator (34)
1	24	0
2–50	600	0
50–250	30	0
250–375	10	2
375–512	0	32

Table 1. Distribution of unmovable pages. Illuminator produces only about 5% hybrid pageblocks compared to Linux.

LIU migration is harmful because it unnecessarily consumes CPU cycles. For example, migrating a baseline page takes about 5 microseconds on our test setup. When a hybrid pageblock contains one unmovable page, the kernel may migrate up to 511 pages from the pageblock adding 2.5 millisecond latency to a huge page allocation. This latency is further exacerbated when many consecutive pageblocks are hybrid. Even worse, huge page allocation can fail even after migrating pages from many hybrid pageblocks.

In addition to copying the page content, compaction also involves other critical operations such as updating the page table(s) and locking a few critical data structures. It also necessitates TLB invalidations when the page being migrated is mapped in the TLB of any CPU core(s). When many hybrid pageblocks are present in the system, LIU migration becomes an unnecessary overhead because *the kernel migrates pages in response to each huge page allocation request without checking the futility of doing so.*

The current kernel design assumes that LIU migration is not an issue as it can prevent long-term fragmentation for sub-pageblock sized allocations. For example, a 64KB allocation request can be served by migrating pages from a hybrid pageblock. It can also free other smaller blocks that can be used to serve future allocations. While we believe such proactive migration is desirable for reducing fragmentation for sub-pageblock allocations, it should not be applicable to THP allocations because it never produces a free pageblock and hence never reduces long-term fragmentation at huge page granularity.

Measuring fragmentation: The overall state of fragmentation depends on the distribution of unmovable pages. Sparsely polluted pageblocks result in higher fragmentation because for a fixed amount of unmovable memory, sparse pollution creates more hybrid pageblocks. Sparse pollution also results in higher compaction overhead as pageblocks with fewer unmovable pages lead to higher amount of LIU migration.

Prior works use FMFI (Free Memory Fragmentation Index) to measure fragmentation [55, 63]. However, FMFI does not consider unmovable pages and hence cannot be used to measure the difficulty of recovering from fragmentation. Instead,

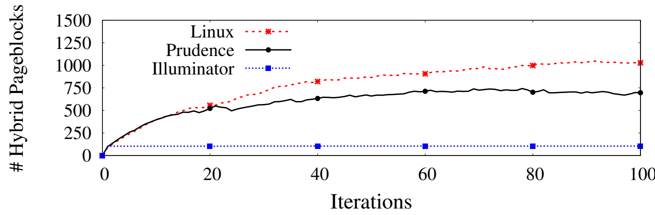

Figure 3. Rate of pageblock pollution with a synthetic benchmark that stresses both the buddy and the slab allocator.

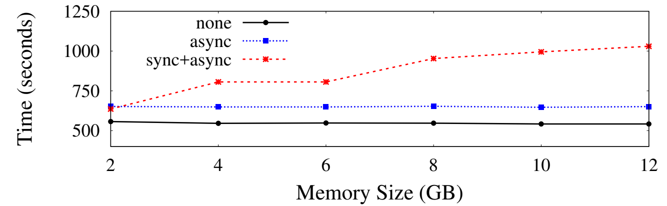

Figure 4. Execution time of `milc` with no compaction (none), with asynchronous compaction (async) and with synchronous and asynchronous compaction (sync+async) at 0.75 unmovability index.

we propose *unmovability index* to measure the degree of fragmentation, which represents the fraction of total pageblocks polluted by unmovable pages. For example, the unmovability index is 0.5 if half of the total pageblocks contain at least one unmovable page each.

4.1.3 Experimental study of Linux fragmentation

We find that fragmentation via pollution and LIU migration are commonplace in Linux. For instance, on a 2GB system which has about 950 pageblocks, the compilation of the kernel source produces 664 hybrid pageblocks with most of the hybrid pageblocks containing a few unmovable pages (see Table 1). In fact, 624 out of 664 hybrid pageblocks have less than 50 unmovable pages each. Interestingly, 24 hybrid pageblocks contain one unmovable page each. As discussed above, sparsely polluted pageblocks lead to significant overhead during memory compaction.

4.2 Delayed reclamation of deferred objects

In the existing slab-based allocators, deferred objects are not reclaimed immediately after the completion of a grace period for several reasons. For example: 1) RCU manages deferred objects in a queue and reclaims their memory by invoking the registered callback function of each object. Thus the reclamation of objects placed towards the end of the queue is delayed, 2) RCU throttles the rate of reclamation of deferred objects to avoid interfering with applications, irrespective of the state of the slab allocator [58], and 3) the kernel threads responsible for reclaiming deferred objects may be preempted. Until reclaimed, deferred objects cannot be reused as they remain invisible to the slab allocator.

Modern applications that perform thousands of update operations per second generate many deferred objects [35]. The delayed reclamation of deferred objects in turn increases slab consumption. It also leads to high slab churns by forcing the slab allocator to populate slab caches, in response to a new allocation request, even if a lot of deferred objects have waited for more than a grace period. Recall that slab pages are unmovable. Hence, unnecessary calls to `alloc_pages` can increase fragmentation via pollution.

Figure 3 shows the impact of fragmentation via pollution with a synthetic benchmark that we use to fragment the memory. Unlike existing tools, our benchmark stresses the buddy allocator (with anonymous mapping) and the slab allocator (with file create and delete operations) at the same time and can be used to control the level of fragmentation. We execute the benchmark in a loop and observe the following.

In Linux, the delayed reclamation of deferred objects increases the number of calls to `alloc_pages` resulting in 1021 hybrid pageblocks at the end of the hundredth iteration. The recently proposed Prudence dynamic memory allocator [28] reclaims deferred objects immediately after the completion of a grace period which reduces the number of hybrid pageblocks to 691, a 35% reduction compared to Linux. Hence, we replace the slab allocator with Prudence.

4.3 Large memory large problems

We also find a counterintuitive side effect of LIU migration i.e., it can make performance worse as the size of memory grows. We observed that at similar unmovability index, the execution time of certain applications increases as the memory size is increased.

To verify this, we measure the performance of `milc` (from SPEC CPU2006) at 0.75 unmovability index with different memory sizes. This workload is chosen because it stresses the memory allocation and compaction code paths due to aggressive memory allocation behavior. Note that a large memory system has more hybrid pageblocks and takes more time to reach a similar unmovability index than a small memory system. However, in this experiment, we do not consider the time taken to fragment the memory; we are interested only in the impact of fragmentation. The amount of LIU migration is also higher on a large memory system as it contains more hybrid pageblocks at similar unmovability index. For example, 2GB and 10GB memory systems have about 750 and 3750 hybrid pageblocks in this case.

Figure 4 shows that `milc` completes in constant time for all memory sizes when compaction is not required i.e., in the non-fragmented case. The execution time is also constant in the fragmented case (but higher than the non-fragmented case) with only asynchronous compaction which is designed

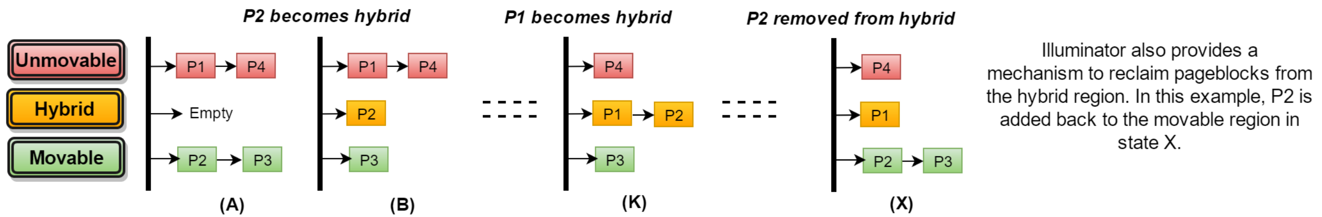

Figure 5. Illuminator improves page clustering by explicitly managing hybrid pageblocks. Once P2 is yielded to the hybrid region during A→B, it is reused until state K. P1 is added to the hybrid region only when P2 fails to allocate memory.

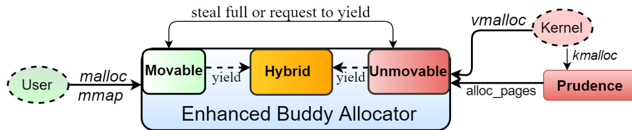

Figure 6. Illuminator explicitly manages hybrid pageblocks in a different region to prevent fragmentation via pollution. Prudence helps Illuminator by minimizing callbacks to `alloc_pages` with timely reclamation of deferred objects.

to avoid interfering with applications. To the contrary, synchronous compaction stalls applications at the time of page fault. Hence, its impact increases as memory capacity (and hence the number of hybrid pageblocks) grows. As a result, 12GB memory system results in 1.9× higher execution time compared to a 2GB memory system.

This occurs because the kernel controls the rate of only asynchronous compaction, with a configuration parameter whose default value is 1.6MB per second. No such limit is enforced on the rate of synchronous compaction whose cost increases as the number of hybrid pageblocks grow. Hence, applications that allocate memory during their entire execution cycle (e.g., `milc`, `bwaves`, `bzip2` from SPEC CPU2006) are heavily impacted by synchronous compaction. Applications that do not allocate memory frequently are not vulnerable to such behavior. We believe a similar issue may be present in Windows as it prohibits making repeated large-page allocations; applications should allocate all large pages one time, at startup [10].

One way to handle this issue is to employ policy-based decisions to also control the rate of synchronous compaction. However, our objective in this paper is to develop an efficient compaction mechanism that can help in both synchronous and asynchronous code paths.

4.4 Impact of fragmentation in virtualized systems

Notice that in virtualized systems, the guest OS (if it is Linux) handles fragmentation with similar algorithms as the host OS. Hence, the severity of fragmentation is higher in virtualized setups because LIU migration takes place in both guest and host when memory is fragmented.

We find that the impact of guest memory fragmentation is more severe than host memory fragmentation as the benefits of huge pages at the guest layer are usually higher than huge pages at the host (for our workloads). However, fragmentation in the host layer also has a significant impact in many cases. Hence, it is important to handle fragmentation in both layers for making the best use of huge pages.

5 Illuminator design and implementation

To efficiently support huge pages, an OS should mitigate fragmentation via pollution and eliminate LIU migration. Intuitively, any solution that minimizes slab memory consumption is also helpful. Illuminator satisfies these requirements by explicitly managing hybrid pageblocks with a simple design. To optimize the memory consumption of slab caches, we use the Prudence memory allocator [28]. Figure 6 represents the high-level design of Illuminator.

5.1 Explicit management of hybrid pageblocks

Illuminator partitions memory in three disjoint sets where unmovable and movable regions serve the same purpose as in Linux while the third region is used to explicitly manage hybrid pageblocks. We represent hybrid pageblocks with *yellow* color. Three-way partitioning based anti-fragmentation provides memory management subsystems a precise view of movability; it guarantees that pageblocks colored red contain only unmovable pages and pageblocks colored green contain only movable pages. This design helps subsystems in making informed decisions.

5.1.1 Mitigating fragmentation via pollution

In Illuminator, the buddy allocator attempts to allocate memory from the corresponding region first (similar to Linux) but prefers hybrid pageblocks for allocation before a region is allowed to steal pageblock(s) from the other. If fallback happens, a pageblock is stolen only if all of its constituent base pages are free. Otherwise, the pageblock is yielded to the hybrid region and its color is updated to yellow. This way, Illuminator ensures that pageblocks are not polluted unnecessarily. The inclusion of hybrid pageblocks does not add performance overhead in the allocation path as they can be accessed by the buddy allocator in constant time.

We explain the memory allocation in Illuminator with an example similar to the one discussed in § 4 (see Figure 5). Illuminator behaves similar to Linux until the system reaches state A as the hybrid region stays empty until this point. If an unmovable page is requested when P1 and P4 have no free pages, the movable region yields P2 to the hybrid region and tags it with yellow color. Illuminator guarantees that a new pageblock is not polluted as long as P2 can successfully serve memory fallbacks. This behavior leads to better clustering of unmovable pages in the long run.

The impact of better page-clustering is shown in Table 1. Illuminator reduces the number of hybrid pageblocks to only 34 compared to 664 of Linux, each with many unmovable pages. Note that almost each hybrid pageblock has more than 375 unmovable pages in Illuminator. It reduces the number of hybrid pageblocks by almost 90% for our synthetic benchmark as well (as shown in Figure 3).

5.1.2 Eliminating LIU migration

Illuminator eliminates THP related LIU migration by decoupling huge page related compaction from the compaction induced by smaller (i.e., sub-pageblock) allocations. This is achieved by avoiding hybrid pageblocks during huge page allocations. The migrate scanner checks the color of each pageblock before migrating pages and skips the entire pageblock range (i.e., 512 contiguous baseline pages) if the pageblock is hybrid. Similarly, the freepage scanner also skips hybrid pageblocks while collecting free pages.

We deliberately allow the freepage scanner to skip hybrid pageblocks so that free space in the hybrid region is not exhausted. This prevents subsequent unmovable allocations from falling back to the movable region (which otherwise may lead to fragmentation via pollution in the long run).

5.2 Reclaiming pageblocks from the hybrid region

A hybrid pageblock may become movable, for example, when memory gets freed and the slab allocator returns all its pages to the buddy allocator. It is important to reclaim such pageblocks from the hybrid region to prevent them from getting polluted by future allocations. Illuminator follows a lazy approach for this task for efficiency; proactively reclaiming pageblocks from the hybrid region necessitates changes in the fast path of the buddy allocator as it requires keeping track of the number of unmovable pages in each pageblock. In performance sensitive code paths, such accounting is considered to be expensive.

Illuminator reclaims pageblocks from the hybrid region during compaction. It queries the number of movable and unmovable pages for each hybrid pageblock encountered by either of the two pointers. If a hybrid pageblock is found to contain only movable (or free) pages, Illuminator updates the pageblock color to green and adds it to the movable region. Similarly, a pageblock is colored red and added to the unmovable region if it is found to contain only unmovable

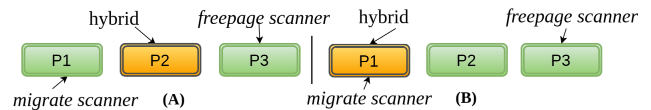

Figure 7. The location of unmovable pages affects the outcome of compaction in the two-way classification approach. In this case, Linux can allocate a huge page only in scenario A while Illuminator can allocate huge page in both A and B.

(or free) pages. If compaction is delayed for a long duration, a thread can be run in the background to periodically scan and reclaim pageblocks from the hybrid region.

5.3 Eliminating susceptibility to page locations

In the Linux kernel, the invisibility of hybrid pageblocks also has an additional side effect i.e., it makes performance susceptible to physical location of unmovable pages. We explain this with an example.

Consider two different scenarios of a system with three pageblocks P1, P2 and P3 along with the positions of the scanning pointers as shown in Figure 7. Assume that half of the baseline pages (i.e., 256 pages) are allocated from each pageblock. Also assume that P2 is hybrid in scenario A and P1 is hybrid in scenario B, with one unmovable page each. In Linux, a huge page can be allocated only in scenario A (by migrating P1's pages to P3). In scenario B, LIU migration of P1's pages to P3 would eliminate the possibility of P2 being allocated as a huge page as it will leave P3 with insufficient space to accommodate the pages of P2.

As a consequence of such behavior, the cost of compaction in Linux depends on the spatial distribution of unmovable pages which in turn leads to variable latency in the allocation of huge pages. Illuminator is not susceptible to such behavior as it avoids hybrid pageblocks during compaction.

5.4 Timely reclamation of deferred objects

We use Prudence to facilitate the timely reclamation of deferred objects. Prudence stores deferred objects in *latent caches* where a latent cache is defined for each slab cache. It also integrates with RCU to provide grace period information to the slab allocator. With such information, the slab allocator reclaims deferred objects immediately after the completion of a grace period.

Integration with the synchronization mechanism provides a complete view of in-use, free and about-to-be-freed objects to the slab allocator. The about-to-be-freed objects also provide crucial hints about the future free operations which are utilized to reduce the footprint of unmovable memory by clustering object allocations within fewer slabs. Together, these optimizations substantially reduce the number of calls to `alloc_pages`. For our synthetic benchmark, Prudence

Benchmarks	Description
milc, mcf, omnetpp, bzip2	Memory and compute intensive applications from SPEC CPU2006 [48]
mummer, tigr	Genome alignment and sequencing applications from BioBench [51]
NPB_CG.D	Congruent gradient algorithm from NAS Parallel Benchmarks [36]
ferret, vips, canneal, bodytrack, x_264	Compute intensive multi-threaded workloads from PARSEC [32]
PostgreSQL, MySQL	Database servers benchmarked with pgbench [15] and sysbench [22] utilities

Table 2. Summary of workloads considered for evaluation.

creates 691 hybrid pageblocks, a reduction of 33% compared to 1021 of Linux as shown in Figure 3.

It is important to highlight that only optimizing the slab allocator is not sufficient because fragmentation via pollution, which is under the control of the buddy allocator, is the primary hindrance to effective OS support for huge pages. However, optimizations in the slab allocator complement the explicit management of hybrid pageblocks by reducing the load on the buddy allocator.

5.5 Implementation notes

Illuminator modifies only the infrequently traversed memory fallback path of the buddy allocator with minor changes in the memory compaction code while Prudence also affects only the deferred free operations. Hence, normal allocation and free operations remain intact. In addition, by eliminating LIU migration, Illuminator alleviates locking overhead on many critical data structures, and minimizes cache pollution and TLB invalidations. Illuminator is about 1500 lines of code change in the Linux kernel 4.5 which affects less than 1.5% of total memory management code.

6 Evaluation

So far we have discussed how Illuminator mitigates fragmentation via pollution with the help of a few examples such as Table 1 and Figure 3. This section presents a detailed evaluation of Illuminator in the context of huge pages.

6.1 Experimental setup and workloads

Our experimental setup consists of an Intel Ivy-Bridge server with 8 cores running at 2.4GHz with 8MB of Last-Level-Cache and a 500GB SSD drive. L1 dTLB and iTLB contain 64 entries each for 4KB pages and 8 entries each for huge pages. The shared L2 TLB contains 512 entries for 4KB pages but does not support huge pages. We evaluate a wide range of workloads summarized in Table 2. Experiments are conducted after disabling swap, to avoid the impact of paging. Experiments that include large memory workloads `NPB_CG.D`, `PostgreSQL` and `MySQL` are conducted with 24GB while other

Terminology	Description
pg_migrate_scanned	Pages scanned by the migrate scanner
pg_free_scanned	Pages scanned by the freepage scanner
pg_migrate_success	Pages migrated during compaction
pg_migrate_failed	Pages that failed during migration
pg_isolated	Pages that were temporarily removed from the buddy allocator

Table 3. Software counters used to measure the cost of memory compaction.

Notation	Description	Value
C_a	Accessing page structure: $sizeof(struct\ page)/word_size$	8
C_{mc}	Migrate page copy: $(C_a + PAGE_SIZE/word_size) \times 2$	1040
C_{sm}	Migrate scanning: C_a, this is equivalent to scanning a page structure	8
C_{sf}	Freepage scanning: C_a, this is equivalent to scanning a page structure	8
C_i	Page isolation: $C_a + W_i$ where W_i is a constant representing locking overhead	28
C_{mf}	Migrate page failure: $C_a \times 2$	16

Table 4. Cost of each activity in the Linux kernel. We take W_i to be 20. However, the cost of compaction is not heavily dependent on its exact value.

experiments are performed with 8GB physical memory. We use the default huge page promotion rate of 1.6MB per second for `khugepaged` as used in Linux distributions.

6.2 The cost model for memory compaction

We use the cost-theoretic model proposed by Gorman to estimate system effort invested in compaction [1]. The model estimates the cost of memory compaction $Cost_{mc}$ in terms of the number of bytes read or written, by tracking system activities and associating each activity with a weight factor (see Table 3 and Table 4). $Cost_{mc}$ is calculated as follows:

$$\begin{aligned} Cost_{mc} &= C_{sm} * pg_migrate_scanned \\ &+ C_{sf} * pg_free_scanned \\ &+ C_i * pg_isolated \\ &+ C_{mc} * pg_migrate_success \\ &+ C_{mf} * pg_migrate_failed \end{aligned} \tag{1}$$

$Cost_{mc}$ determines the amount of memory traffic induced by the compaction code which directly impacts the cache efficiency. Page migration also requires TLB shootdowns as the kernel is generally unaware of whether a page is cached in a TLB (on x86). The cost of TLB shootdowns does not scale well and is known to be a major source of performance overhead in multicore systems [27, 57]. Hence, an efficient solution should minimize $Cost_{mc}$.

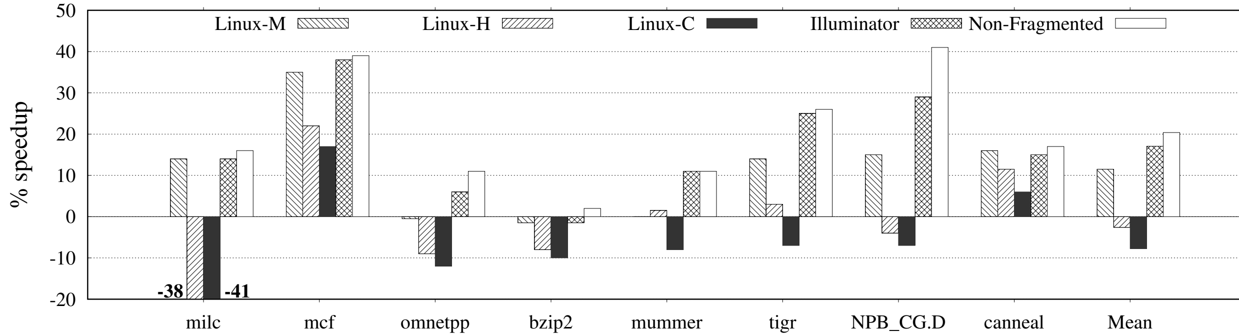

Figure 8. Performance relative to baseline pages at 0.25 (Linux-M), 0.5 (Linux-H) and 0.75 (Linux-C) unmovability indices. Illuminator's performance is presented once which is valid for all fragmentation indices considered. Notice that the performance in Linux degrades as fragmentation increases resulting in worse than the performance of baseline pages at 0.75 unmovability index for most applications.

	At rest		Under stress	
	Linux	**Illuminator**	**Linux**	**Illuminator**
Min	67%	68%	3%	12%
Max	72%	72%	7%	28%
Avg	69%	69%	5%	17%

Table 5. Huge page allocation success rate for `stress-highalloc` which tries to allocate 90% of memory as huge pages.

6.3 Huge page allocations with `stress-highalloc`

`stress-highalloc` [13] is a standard benchmark used by the Linux kernel community developers to quantitatively measure the impact of fragmentation mitigation techniques. The benchmark stresses the memory allocators by running multiple kernel compilation jobs in parallel before requesting 90% of total system memory as huge pages. We run `stress-highalloc` with 3GB memory which is also the recommended size for this benchmark.

Table 5 shows the success rate of huge page allocations as the average of five runs of `stress-highalloc`. Illuminator behaves similar to Linux when the system is at rest (not fragmented) but outperforms Linux under stress (when memory is heavily fragmented with unmovable pages) due to its better management of unmovable pages. Compared to Linux, Illuminator allocates 3.4× more huge pages and reduces $Cost_{mc}$ by more than 80%.

6.4 Performance results on bare-metal

Illuminator delivers benefits when memory is fragmented. To the best of our knowledge, `stress-highalloc` is the only reliable benchmark available for fragmenting memory. However, this benchmark is reliable for only up to 3GB memory which does not provide a realistic context on modern large memory systems. Hence, we create fragmentation with a synthetic benchmark and measure performance at three different stages of fragmentation i.e., *moderate, high* and *critical* which correspond to 0.25, 0.5 and 0.75 unmovability index, respectively. We refer to Linux at moderate, high and critical fragmentation levels as Linux-M, Linux-H and Linux-C.

Illuminator's performance is consistent despite varying fragmentation because it limits the number of hybrid pageblocks to less than 10% of Linux. Hence, less than 8% of total pageblocks are hybrid in Illuminator even when Linux has 75% hybrid pageblocks. Moreover, Illuminator does not allow hybrid pageblocks to participate in compaction. Hence, we present Illuminator results only once as it is valid across all three fragmentation levels considered. We also compare Linux and Illuminator with the best speedup achieved with huge pages in a non-fragmented system.

6.4.1 Overall performance improvement

Figure 8 shows performance relative to baseline pages. In Linux, application performance degrades as fragmentation increases. Notice that *most applications perform even worse than baseline pages in Linux-H and Linux-C.* The worst case is seen for `milc` which suffers 38% and 41% performance loss in Linux-H and Linux-C. Even in cases where Linux improves performance, the benefits are significantly lesser than the non-fragmented case. For example, Linux-H improves the performance of `mcf` by 22% as compared to 39% of the non-fragmented case.

Illuminator outperforms Linux and achieves performance comparable to the non-fragmented system for 5 out of 8 applications while others are within 2–12% of the non-fragmented system. Some performance loss is expected because: 1) Illuminator eliminates only LIU migration; the overhead of normal migration also affects performance, and 2) it takes some time for the kernel to promote huge pages in a fragmented system. Despite these factors, Illuminator provides significantly better performance than Linux in all cases. For example, it improves the performance of `milc` by 55%, `mcf` by 21%, `omnetpp` by 18%, `mummer` by 19%, `tigr` by 32%, `NPB_CG.D` by 38% and `canneal` by 8% compared to Linux-C. The average

Application	Linux-M	Linux-H	Linux-C	Illuminator
milc	17621/34	4657/209	**65/0**	**17594/45**
mcf	121/6	81/4	17/0	135/15
omnetpp	20/90	15/20	17/17	22/101
bzip2	684/12	190/9	110/0	620/55
mummer	112/0	57/0	11/0	483/9
tigr	88/106	74/72	**11/9**	**228/556**
NPB_CG.D	1349/151	890/134	**793/52**	**3405/736**
canneal	92/15	68/11	31/0	92/379

Table 6. Number of huge pages allocated/promoted.

Linux-M	Linux-H	Linux-C	Illuminator
11 (2%)	255 (28%)	343 (37%)	11 (2%)

Table 7. Kernel mode execution time in seconds and the percentage of total time spent in the kernel mode for milc.

performance improvement is also significant i.e., 5.5%, 19.5% and 25% compared to Linux-M, Linux-H and Linux-C.

Next we discuss how Illuminator's better performance is correlated to the number of huge page allocations and the cost of compaction incurred in allocating huge pages.

Huge page allocations: Table 6 shows that Linux's ability to allocate huge pages is impacted by the unmovability index. For example, Linux-M allocates about 17K huge pages to milc which drops to 4.6K in Linux-H and further to only 65 in Linux-C. Illuminator allocates about 17.5K huge pages to milc.

Similarly, khugepaged's ability to promote huge pages in the background also gets impacted by fragmentation in Linux. Notice that Linux-C is unable to promote any huge page for 5 out of 8 workloads. The best case for huge page promotions is observed with tigr and NPB_CG.D where Illuminator promotes 556 and 736 huge pages compared to 9 and 52 in Linux-C.

Compaction overhead: Illuminator significantly reduces the cost of compaction by eliminating LIU migration (Figure 9); the reduction is 10–83%, 19–99% and 62–99% compared to Linux-M, Linux-H and Linux-C.

Efficient compaction in turn reduces CPU utilization and lowers the system time consumed by the applications. While we observed a significant reduction in the system time of all applications, we report it only for milc (see Table 7) as the system time was a small fraction (less than 5%) of the overall execution time for other applications. For milc, system time is higher because this application stresses the compaction code path due to its repeated allocation and freeing patterns. In Linux-M, milc spends only about 2% of its overall execution time in kernel mode which increases to 28% in Linux-H and further to 37% in Linux-C. Illuminator keeps the system time of milc within 2% of the overall execution time.

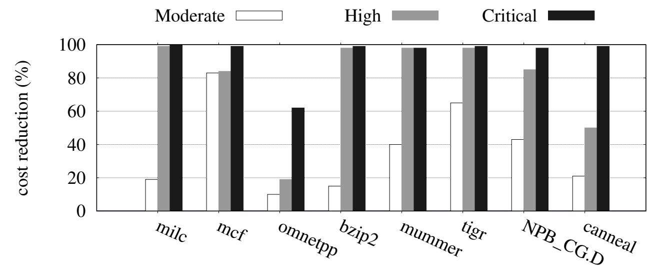

Figure 9. Reduction in the cost of compaction with Illuminator at different fragmentation levels.

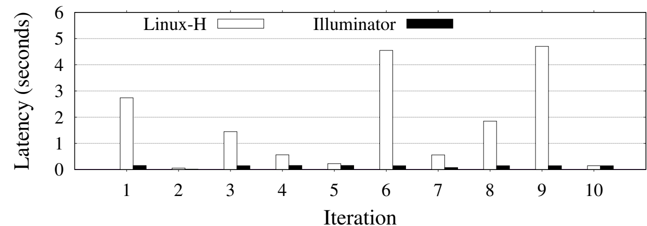

Figure 10. Maximum latency for MySQL read requests from 10 iterations at high fragmentation level.

6.4.2 Latency and OS jitter

We evaluate the impact of LIU migration on latency with the MySQL database server configured with a table of 32 million rows on which a read-only workload was executed with 8 threads. We repeat the experiment 10 times with 30 minutes per iteration and report the maximum latency observed in each iteration (see Figure 10).

In Linux, huge page allocation latency is affected by the location of unmovable pages as the time taken to allocate a huge page depends on the number of hybrid pageblocks encountered by the migrate scanner. Such behavior leads to high latency spikes in Linux. For example in Linux-H, the maximum latency of MySQL read requests varies from 57ms–4702ms across ten iterations. In Illuminator, the maximum latency across iterations varies from 16ms–156ms due to the elimination of LIU migration. Notice that eliminating LIU migration alone does not provide any throughput improvement. When fragmentation via pollution is handled along with LIU migration, we observed 14% higher throughput for MySQL due to high number of huge page allocations.

6.4.3 Performance isolation

As discussed earlier, applications like milc monopolize memory compaction by generating frequent page faults. Note that compaction is a global process that migrates all movable pages encountered by the migrate scanner until it either fails, allocates/promotes sufficient huge pages or gets preempted by the scheduler. In a fragmented system, LIU migration can

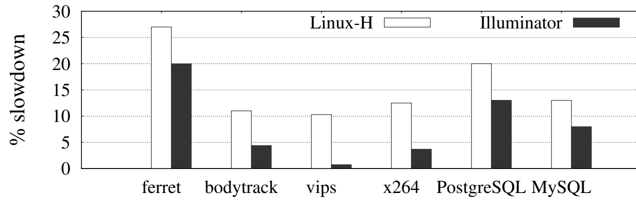

Figure 11. Slowdown for applications (lower is better) while running alongside `milc` at high fragmentation.

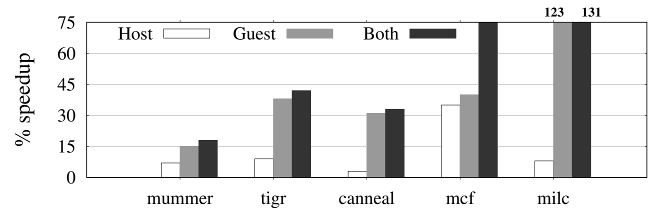

Figure 12. Performance improvement over Linux-H in a virtualized system when Illuminator is deployed at host, guest and both.

lead Linux into a compaction loop which in turn negatively impacts other workloads running on the same machine.

We measure performance isolation by executing a few workloads (one-by-one) alongside `milc` and record the slowdown observed by them in Linux-H and Illuminator. Figure 11 shows the slowdown experienced by the workloads when fragmentation is high. Illuminator provides better performance isolation in all cases. For example, `vips` slows down by 10% in Linux-H but experiences negligible performance loss with Illuminator. Performance isolation is considerably better for all other applications as well in Illuminator. We noticed that Illuminator reduces the number of TLB shootdowns by 63% (for `ferret`) to 90% (for `vips`).

6.5 Performance in virtualized environments

We evaluate a virtualized setup with Ubuntu16.04 running as the host OS with 24GB memory and KVM [45] as the hypervisor. The guest runs with Ubuntu12.04, 8 cores and 8GB memory. In a virtualized system, application performance depends on the state of memory fragmentation at both the guest and the host layers. For brevity, we report results for five applications at 0.5 unmovability index in both guest and host (see Figure 12). Experiments are reported for all the three configurations: 1) when Illuminator is applied only at the host, 2) when Illuminator is applied only at the guest, and 3) when Illuminator is applied at both guest and host.

The performance benefits are higher when Illuminator is applied at the guest as compared to when applied at the host, except for `mcf`, for which the improvement is comparable in both cases. Intuitively, the best performance is observed when Illuminator is operating in both layers. For example, the performance of `mcf` improves by 35% and 41% when Illuminator is deployed at only host or only guest which effectively increases to 75% when it is deployed at both layers. Performance improvement for `mummer`, `tigr` and `canneal` is also substantial i.e., 18%, 42% and 32% in the best case. Similar to the native setup, the best performance improvement is observed for `milc` i.e., 131% as compared to Linux.

In our experiments, guest memory fragmentation results in higher performance loss than host memory fragmentation for two main reasons: 1) our workloads are more sensitive to address translation overhead at the guest, except for `mcf` which is equally sensitive to address translation performance at both layers. As a result, performance loss in Linux due to fragmentation in guest is higher than host memory fragmentation; `mcf` incurs similar performance loss in guest only and host only fragmented scenarios. 2) Applications are executed after fragmenting the guest memory. Hence, the overhead of synchronous compaction performed by the host impacts the process that creates fragmentation; applications, that execute after memory has been fragmented, encounter only the asynchronous compaction of the host. As discussed previously, the performance implications of asynchronous compaction are modest as compared to the synchronous compaction. In contrast, both synchronous and asynchronous compaction of the guest OS impact applications running inside the guest.

7 Related work

OS support for huge pages: Early work discussed multiple ways of supporting huge pages in HP-UX OS [44]. Navarro et al. [49] implemented support for multiple page sizes in FreeBSD with reservation-based huge page allocations and contiguity-aware page reclamation. Carrefour-LP [37] improves performance with huge pages in NUMA systems by balancing the load across memory controllers, maintaining locality and splitting huge pages that cause remote memory accesses. Ingens [63] improves THP support in terms of fairness, bloat and latency by tracking huge page utilization and access frequency of pages. Guo et al. [38] proposed proactively breaking huge pages to improve memory efficiency via page sharing in virtualized environments.

In summary, prior OS research has mainly focused on implementing huge page support, optimizing performance or improving memory efficiency in the presence of huge pages, with different policies applied at the virtual memory layer while fragmentation has received less attention. Illuminator complements policy decisions by making huge page allocations feasible and cost-effective in long-running systems. It can be easily integrated with the techniques discussed above.
Fragmentation mitigation in user space: Allocation and reclamation decisions guided by the size or expected lifetime of objects form the basis for many copying or generational garbage collectors in user space [26, 34, 39, 42, 50, 60]. Such

solutions are not well-suited for mitigating kernel level fragmentation because the kernel has to deal with unmovable pages. Moreover, application specific knowledge (e.g., object lifetime) is, in general, not available to OSs [54, 62].

Fragmentation mitigation in the kernel: S. Kim et al. [59] proposed proactive anti-fragmentation that allocates contiguous memory to each process from the beginning. This way, contiguous memory can be recovered when a process exits or gets killed by the out-of-memory killer. Their solution, however, is specific to Android due to its relatively short-lived processes. Moreover, processes are killed in Android under memory pressure as compared to workstations or servers where paging is preferred over killing a process.

Gorman et al. proposed defragmenting memory with Lumpy reclaim [56]. Contiguity-aware page replacement discussed by Navarro et al. [49] is also similar to Lumpy reclaim. These techniques try to recover memory contiguity by dropping file-backed pages from contiguous regions. Lumpy reclaim was merged in Linux but was removed later as it created regressions due to additional I/O traffic [21]. In current Linux versions, memory compaction is the preferred mechanism for defragmenting memory [6]. We show that compaction leads to severe problems in its current form. Illuminator solves compaction related problems by carefully managing unmovable pages.

Gorman et al. proposed two different anti-fragmentation schemes to aid compaction [55], both of which are merged into Linux. One of the approaches i.e., zone-based demands a static partitioning of memory between movable and unmovable regions at boot time. However, such an approach is difficult to employ when memory capacity is limited or when the workload characteristics are not known in advance. The other approach i.e., Grouping Pages Based on their Mobility type (GPBM [54]) manages the size of memory partitions dynamically and is more suitable for dynamic workloads. Page-clustering algorithm of Linux discussed in this paper is largely influenced by GPBM. We show how page-clustering leads to fragmentation via pollution in its current form, and effectively solve it with Illuminator.

Our previous paper [30] on huge pages is perhaps the closest work to Illuminator, but it has several limitations: 1) it improves page-clustering with $O(n)$ page allocation algorithm (where n is the number of pageblocks), which is unacceptable for large memory systems. In contrast, Illuminator provides $O(1)$ page allocations. 2) The solution relied on the two-way classification of pageblocks whose pitfalls have been discussed throughout this paper. Illuminator shows that cross subsystem visibility of unmovable pages and coordination among several layers (e.g., buddy allocator, slab allocator, and compaction) is crucial. Moreover, Illuminator has also been evaluated across native and virtualized systems with respect to various aspects of performance such as latency, OS jitter and execution speedup.

8 Discussion

Unmovable pages are found in most general-purpose OSs [2, 54]. For example, page tables and DMA pages are typically unmovable [43]. For large memory workloads, page tables can occupy multiple GBs of memory [14] which highlights the importance of efficiently handling unmovable pages. However, different OSs can have different unmovability constraints. OSs that provide support for movable kernel pages (e.g., Windows, Solaris) are less likely to suffer from the issues discussed in this paper. However, not directly mapping memory in the kernel address space leads to a complex design. Hence OSs with movable kernel pages prefer directly mapping certain kernel address spaces in physical memory [11]. For example, FreeBSD reserves a special region in the kernel address space for directly mapping the objects allocated by its slab-like zone allocator. The high-level design of its memory manager is also similar to Linux [3, 52]; it's internal data structures are wired [49] and allocations are handled with the combination of a zone allocator and a buddy allocator. Hence, an interesting future work is to port Illuminator to FreeBSD. However, performance implications depend on how much unmovable memory is present and how the kernel defragments memory. A detailed cross OS study on this topic is a potential future work.

In addition, many prior techniques also demand memory contiguity for improving performance or optimizing energy consumption [29, 41, 46]. Understandably, such techniques are also vulnerable to fragmentation created by unmovable pages. Illuminator can be used to make them more robust in long-running systems.

9 Conclusions

We address the issue of performance regressions caused by huge pages and show how unmovable pages can become a major hindrance to fragmentation mitigation in OS kernels. We propose Illuminator to make huge page allocations feasible as well as cost-effective in fragmented systems, which provides good performance in both native and virtualized systems. While this paper has discussed Illuminator in the context of huge pages, the proposed techniques are generic and can be helpful in mitigating other kernel level fragmentation issues. Illuminator source is available at https://github.com/apanwariisc/Illuminator.

Acknowledgments

We thank the anonymous reviewers, Sorav Bansal, Arpit and Girish Chandrashekar for their constructive feedback. We acknowledge NetApp's generous support for conference travel.

Disclaimer: The views in the article are solely of the authors and not of their employers.

References

[1] A basic model to estimate the cost of memory compaction. https://patchwork.kernel.org/patch/1624461/.

[2] About the Virtual Memory System. https://developer.apple.com/library/content/documentation/Performance/Conceptual/ManagingMemory/Articles/AboutMemory.html.

[3] FreeBSD Manual Pages. https://www.freebsd.org/cgi/man.cgi?query=uma&sektion=9.

[4] Intel Haswell. https://ark.intel.com/products/codename/42174/Haswell.

[5] Intel Skylake. https://ark.intel.com/products/codename/37572/Skylake.

[6] Jonathan Corbet. Memory compaction. https://lwn.net/Articles/368869/.

[7] Jonathan Corbet. Proactive compaction. https://lwn.net/Articles/717656/.

[8] Jonathan Corbet. Virtually mapped kernel stacks. https://lwn.net/Articles/692208/.

[9] khugepaged eating 100% cpu. https://bugzilla.redhat.com/show_bug.cgi?id=879801.

[10] Large-Page Support in Windows. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366720(v=vs.85).aspx.

[11] Mapping physical memory directly. https://www.sceen.net/mapping-physical-memory-directly/.

[12] Mel Gorman. Huge pages part 1 (Introduction). https://lwn.net/Articles/374424/.

[13] MMTests: Benchmarking framework primarily aimed at linux kernel testing. https://github.com/gormanm/mmtests.

[14] Performance Tuning: HugePages In Linux. https://blog.pythian.com/performance-tuning-hugepages-in-linux/.

[15] pgbench. https://www.postgresql.org/docs/9.1/static/pgbench.html.

[16] Recommendation for disabling huge pages for Hadoop. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf.

[17] Recommendation for disabling huge pages for MongoDB. https://docs.mongodb.org/manual/tutorial/transparent-huge-pages/.

[18] Recommendation for disabling huge pages for NuoDB. http://www.nuodb.com/techblog/linux-transparent-huge-pages-jemalloc-and-nuodb.

[19] Recommendation for disabling huge pages for Redis. http://redis.io/topics/latency.

[20] Recommendation for disabling huge pages for VoltDB. https://docs.voltdb.com/AdminGuide/adminmemmgt.php.

[21] Removal of lumpy reclaim. https://lwn.net/Articles/488993/.

[22] sysbench. https://dev.mysql.com/downloads/benchmarks.html.

[23] Tales from the Field: Taming Transparent Huge Pages on Linux. https://www.perforce.com/blog/151016/tales-field-taming-transparent-huge-pages-linux.

[24] The black magic of systematically reducing Linux OS jitter. http://highscalability.com/blog/2015/4/8/the-black-magic-of-systematically-reducing-linux-os-jitter.html.

[25] Why TokuDB Hates Transparent HugePages. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Hadoop_Tuning_Guide-Version5.pdf.

[26] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 261–269, New York, NY, USA, 1990. ACM.

[27] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H Loh. Avoiding TLB shootdowns through self-invalidating TLB entries. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2017*, pages 273–287. IEEE, 2017.

[28] Aravinda Prasad and K. Gopinath. Prudent memory reclamation in procrastination-based synchronization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 99–112, New York, NY, USA, 2016. ACM.

[29] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 237–248, New York, NY, USA, 2013. ACM.

[30] Ashish Panwar, Naman Patel, and K. Gopinath. A case for protecting huge pages from the kernel. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 15:1–15:8, New York, NY, USA, 2016. ACM.

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

[32] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[33] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[34] Darko Stefanović, Kathryn S McKinley, and J Eliot B Moss. Age-based garbage collection. *ACM SIGPLAN Notices*, 34(10):370–381, 1999.

[35] Dipankar Sarma and Paul E. McKenney. Making RCU safe for deep sub-millisecond response realtime applications. In *Proceedings of the 2004 USENIX Annual Technical Conference (FREENIX Track)*, ATC '04, pages 182–191, Berkeley, CA, USA, 2004. USENIX Association.

[36] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks;summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[37] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association.

[38] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. Proactively breaking large pages to improve memory overcommitment performance in vmware esxi. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, pages 39–51, New York, NY, USA, 2015. ACM.

[39] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, September 1975.

[40] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-it-yourself virtual memory translation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 457–468, New York, NY, USA, 2017. ACM.

[41] Heechul Yun, Renato Mancuso, Zheng Pei Wu, and Rodolfo Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, pages 155–166, 2014.

[42] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[43] Ilya Lesokhin, Haggai Eran, Shachar Raindel, Guy Shapiro, Sagi Grimberg, Liran Liss, Muli Ben-Yehuda, Nadav Amit, and Dan Tsafrir. Page fault support for network controllers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 449–466, New York, NY, USA, 2017. ACM.

[44] Indira Subramanian, Clifford Mather, Kurt Peterson, and Balakrishna Raghunath. Implementation of multiple pagesize support in hp-ux. In *USENIX Annual Technical Conference*, pages 105–119, 1998.

[45] Irfan Habib. Virtualization with kvm. *Linux J*, 2008(166), February 2008.

[46] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 178–189, Washington, DC, USA, 2014. IEEE Computer Society.

[47] Jim Mauro and Richard McDougall. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[48] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[49] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan L. Cox. Practical, transparent operating system support for superpages. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11*, 2002.

[50] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *ACM Trans. Program. Lang. Syst.*, 27(6):1097–1146, November 2005.

[51] K. Albayraktaroglu, A. Jaleel, Xue Wu, M. Franklin, B. Jacob, Chau-Wen Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, ISPASS '05, pages 2–9, Washington, DC, USA, 2005. IEEE Computer Society.

[52] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[53] Mel Gorman and Patrick Healy. Performance characteristics of explicit superpage support. In *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA '10, pages 293–310, Berlin, Heidelberg, 2012. Springer-Verlag.

[54] Mel Gorman and Patrick Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 41–50, New York, NY, USA, 2008. ACM.

[55] Mel Gorman and Andy Whitcroft. The what, the why and the where to of anti-fragmentation. In *Linux Symposium*, page 369–384, 2006.

[56] Mel Gorman and Andy Whitcroft. Supporting the allocation of large contiguous regions of memory. In *Linux Symposium*, page 141–152, 2007.

[57] Nadav Amit. Optimizing the TLB shootdown algorithm with page access tracking. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '17, pages 27–39, Santa Clara, CA, USA, 2017. USENIX Association.

[58] Paul E McKenney, Dipankar Sarma, Ingo Molnar, and Suparna Bhattacharya. Extending RCU for realtime and embedded workloads. In *Ottawa Linux Symposium, pages v2*, pages 123–138. Citeseer, 2006.

[59] Sang-Hoon Kim, Sejun Kwon, Jin-Soo Kim, and Jinkyu Jeong. Controlling physical memory fragmentation in mobile systems. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 1–14, New York, NY, USA, 2015. ACM.

[60] Tamar Domani, Elliot K. Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 274–284, New York, NY, USA, 2000. ACM.

[61] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 25–35, New York, NY, USA, 2016. ACM.

[62] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. Application level ballooning for efficient server consolidation. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 337–350, New York, NY, USA, 2013. ACM.

[63] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, GA, 2016. USENIX Association.