A Case for Protecting Huge Pages from the Kernel

A Thesis

Submitted for the Degree of

Master of Science (Engineering)

in the **Faculty of Engineering**

by

Naman Patel

Dept. of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

JULY 2016

DEDICATED TO

To my Late Grandfather, Parents, Grandmother and Teachers

Signature of the Author: .....

Naman Patel
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

Signature of the Thesis Supervisor: .....

K. Gopinath
Professor
Dept. of Computer Science and Automation
Indian Institute of Science, Bangalore

Acknowledgements

As I write this thesis, I can distinctly recall my experiences in Indian Institute of Science which includes the happiness of heavens and the sorrows of graveyard. The happy times will serve as good memories while the sorrows turned out to be much needed learning.

I will like to take this opportunity to thank my idol and research adviser Prof. K. Gopinath who has been instrumental in me being able to contribute to the core area of Operating Systems. His guidance and support was very crucial during the tough times of program. From changing problem definitions to the constructive discussions I am indebted to the path he allowed me to follow.

This thesis is dedicated to my late grandfather who taught me how to live, who taught me how to walk and stand up when I fall. If not for him, I would not have even entered the gates of IISc. I am also grateful to my parents and my cousins who encouraged me to work hard.

I would like to thank my friend Ashish Panwar for helping me out. If not for his help then I would have struggled to get the desired results this quickly.

And the journey would not have been enjoyable without having friends by my side. Amit, Pallavi, Rashmi, Soniya and Suvita (monotonically ascending) were there always, no matter what. The gossips on the stairs, birthday celebrations and the regular fights were the building blocks of the friendship that we shared.

I will also like to thank my lab mates Ashish, Sah, Lal, Sandeep, Poorna, Milan, Priyanka, Toms, Abhinav, Abhishek, Rupesh, Manoj Sir, Yash, Puneet, Priyank, Ashwin, Fazal, Jawad, Srimithra, Arpith, Shail, Vivek who helped me through out my stay here in CASL. They always kept the environment of the lab pleasant and stress free. All the games that we played, snacks at prakruthi, parties and discussion will be cherished through out the rest of my life.

I am also indebted to my B.E. friends Devendra, Prashray, Tushar, Chirag, Nisarg, Maunik, Deepa and Madhuri who directly or indirectly pushed me towards my goal. They motivated me to push the limits and explore the unexplored. My B.E. teachers Radha Teredesai, Aekta Shah, Saurabh Shah provided the necessary stepping stone for the journey that eventually took me to IISc.

Publication based on this thesis

'A Case for Protecting Huge Pages from the Kernel' in *7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 4-5 Aug, 2016)*

Abstract

Modern architectures support multiple size pages to facilitate applications that use large chunks of contiguous memory either for buffer allocation, application specific memory management, in-memory caching or garbage collection. Most general purpose processors support larger page sizes, for e.g. x86 architecture supports 2MB and 1GB pages while PowerPC architecture supports 64KB, 16MB, 16GB pages. Such larger size pages are also known as superpages or huge pages. With the help of huge pages TLB reach can be increased significantly. The Linux kernel can transparently use this huge pages to significantly bring down the cost of TLB translations. With Transparent Huge Pages (THP) support in Linux kernel the end users or the application developers need not make any modification to their application.

Memory fragmentation which has been one of the classical problems in computing systems for decades is a key problem for the allocation of huge pages. Ubiquitous huge page support across architectures makes effective fragmentation management even more critical for modern systems. Applications tend to stress system TLB in the absence of huge pages, for virtual to physical address translation, which adversely affects performance/energy characteristics in long running systems. Since most kernel pages tend to be unmovable, fragmentation created due to their misplacement is more problematic and nearly impossible to recover with memory compaction.

In this work, we explore physical memory manager of Linux and the interaction of kernel page placement with fragmentation avoidance and recovery mechanisms. Our analysis reveals that not only a random kernel page layout thwarts the progress of memory compaction, it can actually induce more fragmentation in the system. To address this problem, we propose a new allocator which takes special care for the placement of kernel pages. We propose a new region which represents memory area having kernel as well as user pages. Using this new region we introduce a staged allocator which with change in fragmentation level adapts and optimizes the kernel page placement. Later we introduce Illuminator which with zero overhead outperforms default kernel in terms of huge page allocation success rate and compaction overhead with respect to each huge page. We also show that huge page allocation is not a one dimensional

problem but a two fold concern with how the fragmentation recovery mechanism may potentially interfere with the page clustering policy of allocator and worsen the fragmentation.

Our results show that with effective kernel page placements the mixed page block counts reduces upto 70%, which allows our system to allocate 23% more huge pages than the default Kernel. Using this additional huge pages we show up to 41% improvement in terms of energy consumed and reduction in execution time up to 40% on standard benchmarks.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Complex and large working sets of modern applications are known to gain substantial benefits with huge pages. As a result, robust huge page support is available across general purpose processors at different granularity. For example, x86_64 systems support 2MB and 1GB huge pages while PowerPC facilitates huge pages of 64KB, 16MB and 16GB. When the working set of an application surpasses system TLB reach, a large proportion of system effort goes into translating virtual to physical addresses. While the memory capacity in systems has grown exponentially the TLB entries available has not increased similarly. With increase in memory availability, the application working set has also gone up. This has significantly widened the memory to TLB reach ratio which means many of the pages referenced by applications incurs a TLB miss which increases the translation cost by many folds. Huge pages can minimize this translation cost by mapping large portions of process address space into a single TLB entry. Further, huge pages can improve cache prefetching as now the prefetching need not be stopped at 4KB page boundary. Also once the system is using huge pages, the page table walk cost reduces due to lesser number of page tables required to map the same amount of memory as show in Figure 1.1.

Linux kernel developers have made significant effort to make huge pages available and be automatically used by applications with Transparent Huge Page(THP) support. THP is an efficient way of using huge pages for the backing of virtual memory with huge pages that supports the automatic promotion and demotion of page sizes. THP maximizes the usefulness of free memory compared to the reservation approach of hugetlbfs by allowing all unused memory to be used as cache or other movable (or even unmovable) entities. THP allows paging and all other advanced virtual memory features to be available on the huge pages. Also no modifications are required to application for making use of THP.

However, facilitating allocation of large contiguous blocks is a challenge for operating sys-

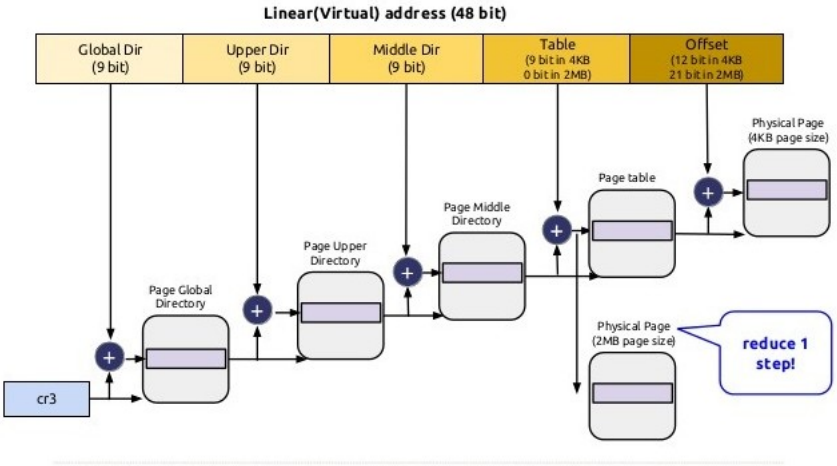

Figure 1.1: Huge page in 64 bit Linux for x86_64

tems because of fragmentation.

Fragmentation, which is mainly categorized in two forms; internal and external, spans across all forms of storage/memory domains (e.g., disks, process heaps, physical memory). External fragmentation is a situation where sufficient free resources are available but allocation request can not be satisfied because of the non-contiguity of free regions. Internal fragmentation occurs when a larger block than the request size is allocated. Both internal and external fragmentation leads to wastage of space but apart from that there are many other implications. It may also incur performance/energy loss due to inefficient resource utilization like huge pages. In the context of this thesis, we only talk about external fragmentation in physical memory at huge page granularity (i.e., unavailability of free memory regions aligned with huge page size) and refer to it as fragmentation for simplicity.

Defragmentation (also known as compaction) is a popular mechanism to recover from fragmentation in running systems. However, there are subtle differences in defragmenting physical memory as compared to disks or process heaps. In the latter cases it is relatively easy to copy objects from one place to another while physical memory can have significant number of unmovable pages. Operating systems typically have a large pool of non-pageable kernel memory objects which can neither be swapped to disk nor migrated in memory primarily due to the lack of unmanaged references. Pages backing these objects are pinned in memory during their lifetime and it is the placement of these pages on physical address space that decides the degree to which compaction can accumulate large contiguous regions. As most of the operating systems typically have unmovable kernel pages [14], compaction is limited by these kernel pages since it now can only move user pages.

Every kernel page, during its lifetime, prevents its memory block from being allocated as huge page to user applications. Hence, facilitating huge page allocations in the presence of unmovable pages demand different solutions for controlling fragmentation. It requires protecting contiguous blocks (that constitute huge pages) from kernel memory allocations. Linux utilizes *page clustering* based anti-fragmentation [10] to confine unmovable allocations within fewer regions. It helps memory subsystem in two ways: delaying fragmentation and facilitating easy recovery from it. However, page clustering suffers badly in its current form when memory gets divided in smaller blocks. We show that it happens due to poor page selection in an infrequently traversed path of underlying page allocator.

Interestingly, minimizing the number of blocks occupied by kernel pages is not sufficient. Our experiments show that physical location of kernel pages also impact memory subsystem and their interference with compaction can actually exacerbate fragmentation in long running systems.

In this thesis we explore 4 different approaches to mitigate the gap between the policy and implementation of page clustering anti fragmentation approach used by Linux kernel. We modify the virtual memory subsystem of Linux kernel so as to adapt to the varying level of fragmentation which in turn allows us to make an intelligent choice of kernel page selection. We also propose Illuminator which introduces a new region which ensures better placement of kernel pages and easy recovery from fragmentation. This ultimately helps in reducing the external fragmentation caused due to misplacement of kernel pages and boost the system ability to allocate more huge pages to applications.

The primary contributions of this thesis are:

1. Evaluating the gap between policy and implementation of page clustering in Linux kernel
2. Evaluating the impact of different kernel page layouts on memory compaction.
3. Propose and implement an adaptive allocator which monitors the fragmentation and invests effort accordingly for kernel requests
4. Propose and implement Illuminator, which introduces a new region called composite region which enables easy allocation and recovery of huge pages. It also reduces the overhead on compaction when called for accumulating huge pages.
5. Performance analysis of the proposed allocator with respect to fragmentation, huge pages allocation, execution time and power savings

The rest of this thesis is organized as following : Chapter 2 provides overview of Linux memory manager, buddy allocator and issue related fragmentation and compaction. Chapter 3 discusses related work. Chapter 4 provides the design and implementation of our framework. Results related in the context of memory fragmentation, huge page allocation success, TLB miss reduction, execution time reduction with energy savings are discussed in Chapter 5 along with the detailed evaluation of system performance with modified VM framework. Finally, we present our conclusion and directions for future work in Chapter 6.

Chapter 2

BACKGROUND

This chapter discusses a subsystem of Linux virtual memory framework which manages the physical memory and we outline the page allocation and page free operations. We modify this part of the Linux VM to deal with the fragmentation arising due to the placement of kernel pages.

2.1 Overview of Linux Memory Manager

Currently the Symmetric Multiprocessing (SMP) systems comes with processors that provide local memory for improving the system performance in Non-Uniform Memory Access (NUMA) architecture. Remote node memory access latency can be significantly higher based on the distance between requesting CPU and memory. Commercial applications can perform poorly upto 20% due to remote access. Operating systems generally are NUMA aware which try to optimize the performance through allocating the memory from local node.

Linux divides the memory nodes into different groups called as memory zones. The standard Linux kernel zone layout on x86 and x86_64 architectures are shown in Figure 2.1. The first 16MB of physical memory is represented by Zone DMA while Zone NORMAL is represented by the portion of memory which can be directly mapped in kernel address space. 4 GB of address space is typically divided in the ratio of 3GB and 1GB between user and kernel space respectively. Kernel reserves 128MB for the mappings beyond 1GB and due to this the zone NORMAL represents memory upto 896MB. The remaining memory if availbale is given to zone HIGHMEM.

Zone HIGHMEM is not at all required in 64-bit architecture as most of the memory can be directly mapped in the kernel address space and DMA32 occupies upto 4GB of memory which can be addressed by modern DMA devices.

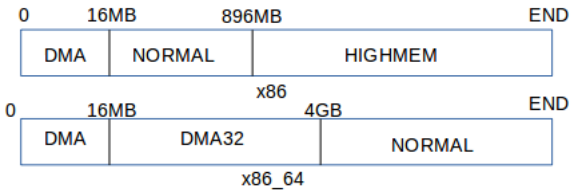

Figure 2.1: Standard Linux Kernel Zone Layout

Linux uses a variant of binary buddy allocator for managing page allocation and page freeing operations as shown in Figure-2.2. Most commonly used page size across different architectures and operating systems is 4KB [1]. It maintains MAX_ORDER (defined as 11 in x86) doubly linked lists. The data structure used to represent this list is called *free_area* in kernel source. Each list, from order 0 to MAX_ORDER-1, represents 2^{order} contiguous pages in a single list entry [10]. While allocating a page, buddy allocator searches a desired list and allocates first page from it. A higher order page can be split into multiple smaller pages to satisfy a memory allocation request if the list corresponding to required order page is empty. While freeing a page, buddy allocator first checks if the neighboring page (also called as buddy) of the same size is free. The pages are merged into a larger page if the buddy is found to be free before inserting into the free list. The process is repeated until either the buddy is not free or the free pages have already been merged to form the largest free page.

Pageblock is an order 9 page, which contains 512 contiguous 4KB pages. Pageblock is aligned with default huge page size of x86_64 i.e., 2MB. Each pageblock has a migrate type associated with it which represents the type of pages present in that pageblock. Allocations of similar type are tried to be clustered inside a single pageblock. The pageblock migrate type is also used to decide which list the freed page should be added.

Despite having both internal and external fragmentation problems, buddy allocator is preferred for its speed [10]. It can allocate even large chunk of contiguous pages in a single look-up on *free_area* structure as long as free memory is available. Internal fragmentation occurs when the granularity of memory allocation is larger than the object size requested. External fragmentation is a situation where sufficient memory is available for allocation but is split in multiple blocks (non-contiguous) and hence can not be used. Slab allocator handles internal fragmentation carefully which is used to allocate small objects. External fragmentation is somewhat mitigated by eliminating the need for large contiguous chunks of physical memory from different Linux subsystems and by providing virtually contiguous memory to process. However, larger blocks are needed in allocation of huge pages which require contiguous area in physical address

[1] Larger than 4KB page size are also known as huge page or super pages

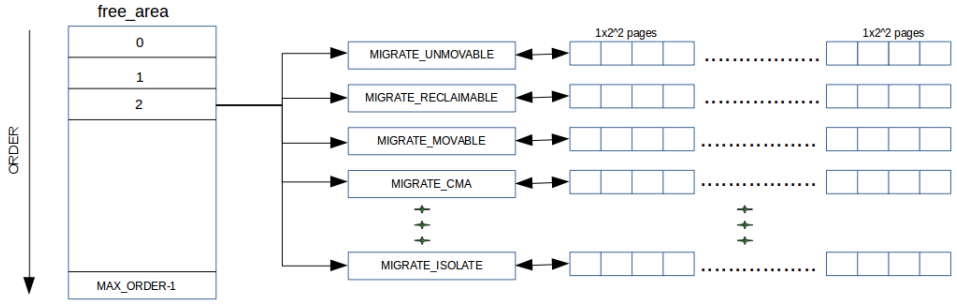

Figure 2.2: Layout of *free_area* structure. Anti-fragmentation lists shown correspond to an order 2 block and hence contain 4 pages in a single block.

space.

One important factor contributing to external fragmentation is the presence of non-movable pages, pages that are pinned in memory and can not be moved to another physical location using page migration. Generally these pages are owned by kernel and contain critical data like process page tables or driver data. For user pages, there is a back trace from a page to the set of page tables that are referring to it and migrating such a page is possible by copying page content and modifying those page table entries. Kernel pages are accessed by kernel-space pointers rather than page tables and hence can not be migrated without modifying these pointers for which there is no back trace as of now.

When system runs out of large contiguous memory blocks and memory allocations start failing, zone compaction is performed to cluster free pages together by migrating some pages from one location to another. However, page migration is often restricted when non-movable pages scatter throughout entire address space and leave very few (or none) contiguous blocks with only movable pages.

2.2 Anti Fragmentation

To deal with the problem of non-movable pages, each order's free list is divided in different sub-lists as part of anti-fragmentation [10] which is a technique that tries to stop fragmentation from happening in the first place. These sub-lists are managed by *free_list* structure which is embedded inside *free_area*. Figure 2.2 shows layout of *free_area* structure including relevant sub-lists for order 2 free list. MIGRATE_UNMOVABLE and MIGRATE_RECLAIMABLE typically represents memory used by kernel with one major distinction that UNMOVABLE pages can not be easily moved or reclaimed while RECLAIMABLE are pages which kernel can directly reclaim (such as those used by slab caches).

MIGRATE_MOVABLE is associated with user applications and contain anonymous or page

Algorithm 1 : Kernel Memory Fallback Routine

```
1:  fallback_kernel(request_order)
2:  for order = 10 to request_order do
3:      if !list_empty(USER, order) then
4:          page = get_head_page(USER, order)
5:          break
6:      end if
7:  end for
8:  if page then
9:      nr_reserved = reserve_page_block_pages(page)
10:     if nr_reserved >= page_block_nr_pages/2 then
11:         change_page_block_domain(page, KERNEL)
12:     end if
13:     /* split if page is large, before returning */
14:     return page
15: else
16:     return NULL
17: end if
```

cache pages that can be migrated any time. MIGRATE_CMA (CMA stands for Contiguous Memory Allocator) is for DMA devices that can take advantage of large contiguous blocks and need not go through the complex set of structures to access memory pages. One difference between CMA and MOVABLE migrate types is that migrate type of a pageblock containing MIGRATE_CMA pages cannot be changed explicitly. Pages belonging to MIGRATE_ISOLATE free lists are not used for allocation. It serves the purpose of temporarily removing pages from the allocation path and is often required in cases like NUMA migration. There are a couple of more lists as well but they are of no interest in the context of this thesis.

Another solution available in Linux for controlling fragmentation is the presence of optional ZONE_MOVABLE [11]. It creates a zone that is usable only for movable allocations (anon/page-cache pages). Size of this zone is determined by the *movablecore* (or *kernelcore*) parameter specified at boot-time. Similar to ZONE_MOVABLE, we also consider everything to be unmovable except MIGRATE_MOVABLE and MIGRATE_CMA pages.

2.2.1 Fallbacks

When an allocation request comes in, kernel knows its migrate type based on the source of allocation for example the page fault, and allocates pages from the respective list. It can also steal pages (called as *fallback* in kernel source) from another list if no pages are present in the requested lists of same or higher order, we will refer to this event as fallback in future. The trick is to steal the highest order available page (preferably 4MB) from the closest migrate type.

Stolen pages from a higher order list are used to satisfy subsequent requests of same type which helps in clustering same migrate type pages together. Once non-movable pages are clustered in large blocks, compaction can provide large free pages by migrating pages from movable memory blocks.

2.3 Memory Compaction

When an allocation request fails, Linux kernel checks the fragmentation index to deduce if the failure was due to low memory or due to external fragmentation. If the failure is due to low memory then page caches are dropped and the allocation is tried again else if the failure was due to external fragmentation than it calls memory compaction to compact physical memory so as to make contiguous space for the allocation to be satisfied. If kernel pages are clustered in a few page blocks, user pages can be moved around in memory relatively easily to enable huge page allocations. For the purpose of understanding, compaction can be thought of as a combination of two scanners:

1. The first one starts at the bottom of memory zone and prepares a list of in-use (but movable) pages. We refer to them as source pages hereafter.

2. The other works at the top of memory zone and collects free pages in a linked list which we call as target pages.

Once the two scanners meet, source pages are copied onto target list and their references are updated properly. As shown in figure 2.3 before the compaction there was external fragmentation due to which even order 1 (2 consecutive pages) request will fail, but after compaction, even order 2 (4 consecutive pages) requests can be satisfied.

2.4 Experimental Study

Compilation of Linux kernel is popularly known for fragmenting physical memory. Hence, we run kernel builds for analyzing Linux fragmentation management framework on a 4GB system.

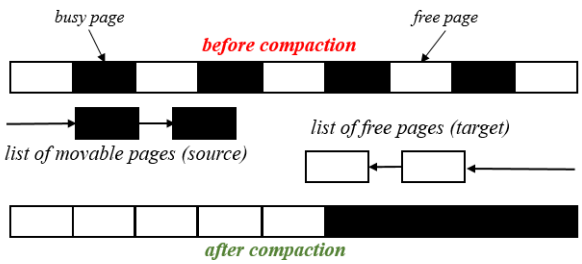

Figure 2.3: Memory Compaction

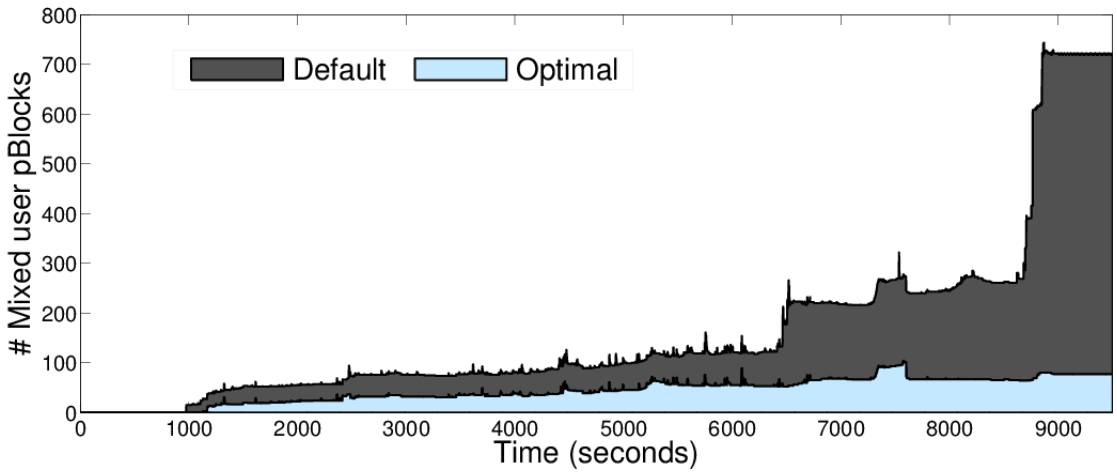

Figure 2.4: The mixed page block formation rate with default and optimal algorithms on a 2GB system.

A detailed analysis of our experiments is discussed below:

2.4.1 Issues with Anti-Fragmentation

Anti-fragmentation works well in terms of restricting kernel memory within few page blocks as long as kernel memory is being allocated from its respective domain or fallbacks are able to reserve relatively large page blocks. However, its behavior is not sustainable in busy systems running for long periods of time for following reasons:

1. A random page block selection during fallbacks is likely to result in insufficient pages getting reserved especially when free pages are not available in high order buddy lists. It causes consecutive kernel allocations to fallback on different user page blocks.

2. When fallbacks reserve pages without updating the page block ownership (i.e., *if* condition fails on line 10 in Algorithm-1), page block gets divided between two domains. On one hand, its free pages are allocated as kernel memory while pages that were already allocated from it are freed back to user domain.

Such behavior of kernel memory fallbacks can cause a large number of page blocks containing kernel pages in long run. Note that none of the page blocks containing kernel pages can be used as huge page by user applications irrespective of the amount of free memory available in them. Hence, we can measure the impact of kernel page placement on memory fragmentation by calculating the number of tainted page blocks. A page block is called tainted if both kernel

pages and user pages have been allocated from it. Figure-2.4 shows the number of tainted page blocks for a single kernel build. Initially anti fragmentation approach works well but with time the number of tainted page blocks shoots up in case of default kernel whereas our optimal kernel (explained in Section 4.1) is able to limit the tainted page block significantly. This shows that the anti fragmentation approach is prone to scatter kernel pages on physical address space thereby greatly reducing the possibility of huge pages allocation to user applications.

Further, we define for each user page block the degree of pollution as the fraction of its memory allocated as kernel pages. Table-2.1 shows the degree of pollution for all user tainted page blocks at the end of two successive kernel builds. We find more than 80% (1345 out of 1667) user page blocks being mixed while majority of them are mixed because of the presence of very few kernel pages (24 page blocks contain only 1 and more than 90% of page blocks contain less than 10% kernel pages).

2.4.2 Issues with Memory Compaction

Memory compaction, in its current form, does not interact with buddy allocator[1] while collecting target pages. It scans memory from the upper end and prepares a list of free pages irrespective of memory domains. Our observations indicate that its ignorance of anti-fragmentation is actually a source of exacerbated fragmentation in long running systems for following reasons-

- If source pages are selected from page blocks containing kernel pages, free regions will not be accumulated at huge page granularity.
- If target pages are selected from page blocks belonging to kernel domain, it will constrict free space in kernel domain. It is more likely to cause future kernel memory requests to fallback on user page blocks.

[1]Note that buddy allocator can not help much when most page blocks contain kernel pages. It is the primary reason why compaction does not get target pages via buddy allocator.

Degree of Pollution	**# pageblocks**
<1	112 (8%)
1-2	89 (7%)
2-4	241 (18%)
4-10	781 (58%)
>10	121 (9%)
>25	4
>40	0

Table 2.1: Distribution of kernel pages across all user page blocks. Note that more than 90% page blocks are less than 10% polluted with kernel memory.

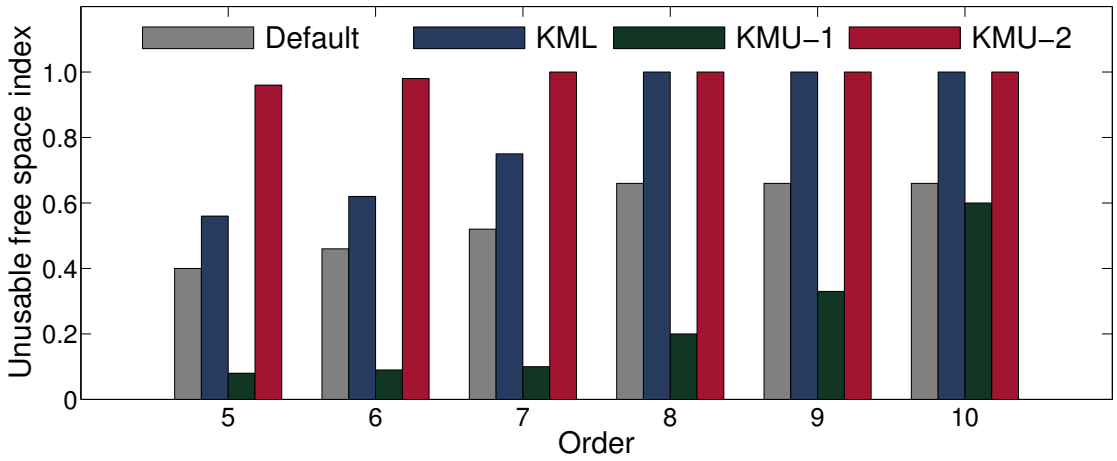

Figure 2.5: Unusable free space index (lower is better) with different kernel memory arrangements.

To investigate and validate the properties discussed above, we use two modified kernels i.e., KML (Kernel Memory Lower) and KMU (Kernel Memory Upper) for placing kernel pages towards the lower and upper end of memory respectively and measure the effectiveness of memory compaction. We expect following behavior from the two variants-

- In KML, kernel pages are aligned towards the lower end of memory while page blocks towards the upper end belong to user applications. Although pages can be allocated arbitrarily under memory pressure (depending on the state of free memory), bulk of kernel pages gets placed towards the lower end by default. Hence, compaction can not produce free huge pages since source pages get selected from page blocks containing kernel pages in KML.

- In KMU, since source pages are selected from user page blocks, it should be relatively easy to free contiguous regions for immediate use. However, target pages come from kernel page blocks which should result in exacerbated fragmentation even for a few kernel memory allocations after compaction.

We can measure the success of memory compaction using *unusable free space index* [10], for different memory block sizes as follows -

$$F_u(j) = \frac{TotalFree - \sum_{i=j}^{max} 2^i k_i}{TotalFree} \tag{2.1}$$

where *TotalFree* is the number of free pages in system, j is the order of desired memory block size, *max* is the largest allocation order and k is the number of free blocks of order i. $F_{\mathrm{u}}(j)$ essentially represents the fraction of free memory that cannot be used to satisfy a memory request of order j. For example, a value of 0 means that any free block can be used for a particular request and hence a lower value is desirable for $F_{\mathrm{u}}(j)$.

We invoke full memory compaction via *proc* interface after finishing kernel compilation with KML, KMU and default kernel and measure the *unusable free space index*. While KML was not able to accumulate any block greater than order 7, KMU performed better than the default kernel. We monitor the KMU configuration a little longer after compaction is finished to observe its long term consequences. Though KMU performed better than the other two variants at the end of memory compaction, the *unusable free space index* reaches almost 1 for all memory blocks (greater than order 5) in a short period of time which is worse than the other two from a long term perspective. Figure-2.5 shows the *unusable free space index* across all configurations. Note that KMU-1 and KMU-2 represents the KMU configuration immediately and sometime after compaction. The experiments confirm our observations that the physical location of kernel pages plays an important role in determining the success of memory compaction.

Another observation come from a different configuration of KML where we forced kernel memory to fall aggressively towards the lower end of memory using page migration. We observed that despite tainted page block count being very low, the *unusable free space index* for huge pages was still very high at the end of our test. It indicates that minimizing the number of tainted page blocks is necessary but not a sufficient condition for controlling fragmentation.

2.5 Summary of Motivation for Thesis

The complex interconnection among memory compaction, anti-fragmentation and the layout of kernel pages makes fragmentation a challenging problem. While effective anti-fragmentation and defragmentation are of paramount importance within their disciplines, a cooperative fragmentation management is required for a robust huge page friendly memory manager.

Fragmentation limits the use of huge pages for applications which in other case would have improved the execution time and energy performance of this applications. As of now Linux does not invest any effort to reduce the fragmentation due to kernel page placement under stressful conditions.

The real problem with respect to fragmentation appears when kernel memory starts growing beyond its domain. When this happens, kernel pages need to be allocated from user domain (we call this event as memory fallback in the rest of the thesis). During fallback, a random page is selected from the highest order non-empty buddy list and rest of the pages from its page

block are stolen for future kernel memory allocations. However, a random page selection results in an inefficient realization of page clustering causing interleaving of kernel and user pages on majority of page blocks in long run. Compaction also suffers due to this interleaving as page blocks containing kernel pages can not be freed entirely. Over a period of time, system finds it difficult to provide applications with huge pages.

In this thesis we focus on improving the anti-fragmentation framework such that the fragmentation due to kernel page placement is minimized with minimum effort.

Our analysis shows that fallback is a rarely executed code path and hence there exists sufficient scope to optimize its impact on fragmentation without sacrificing the overall performance of memory allocator.

Chapter 3

RELATED WORK

Several fragmentation management schemes have been developed over the years which tries to reduce the fragmentation caused in physical memory. Most of these solutions have gone with separation of kernel memory and user memory into some kind of regions or domains. Some have come up with smaller granularity separation for user and kernel memory which is flexible while few approach follows static boot time separation. Internal fragmentation is a concern when we fix the size of any domain. Thus there is a trade-off which these techniques try to balance. Most of these frameworks have been explored and analyzed in Linux kernel.

Gorman et al. [10] introduced two approaches toward active anti-fragmentation. The list method tries to maintain different lists at each order to allocate requests coming from kernel or user. If the request cannot be satisfied from the designated list then it can steal from other list. Other method is to separate allocations into different zones. At boot time the user can specify the amount of memory that should be reserved for kernel allocations. This is a static approach as the memory to be reserved has to be mentioned beforehand and clearly, specifying a suitable value is difficult. Authors show that the list based and zone based techniques are able to reserve huge pages after rigorously stressing the memory, but the zone based technique is able to out perform the list. Trade-off between flexibility and performance is discussed which points that list based is more suitable approach for current memory systems but it is the zone based technique which gives best performance.

Gorman et al. [11] propose a memory compaction mechanism that migrates pages from sparsely populated to dense regions when enough memory is free to avoid reclaiming pages. Authors highlight that parallel allocators prevent contiguous allocations by taking free pages from regions being reclaimed. They also propose a method for addressing this by making pages temporarily unavailable to allocators. Both the papers by Gorman are considered to be practical and hence they are incorporated in Linux as a primary countermeasure for memory

fragmentation. However, the anti fragmentation or the page clustering policy mainly focuses on ensuring physical contiguity of small-order pages and is insufficient for higher order pages like the huge pages.

Gorman [9] proposed a scheme to group pages based on their mobility type to facilitate easy recovery of huge pages in a fragmented system. Gorman also proposed a contiguity aware page reclamation algorithm to free memory from huge page aligned regions. While the approach works well when fragmentation is low, it fails to recover huge pages once the system is severely fragmented.

Sang-Hoon Kim et al. [13] presented a proactive anti-fragmentation approach that groups pages with the same lifetime, and stores them contiguously in fixed-size contiguous regions. In Android Linux when a process is killed to secure free memory, a set of contiguous regions are freed and subsequent contiguous memory allocations can be easily satisfied without incurring additional overhead. The proposed scheme greatly alleviates fragmentation, thereby reducing the I/O buffer allocation time, associated CPU usage, and energy consumption. The work is very specific to Android as only in mobile space does the process is killed when memory needs to be reclaimed as there is no swap space for swapping pages. Hence, while this solution is quite effective in Android Linux it does not address fragmentation occurring in server and workstations. Navarro et al. [17] proposed a reservation based scheme to support huge pages. They claim that fragmentation must be in a controlled to benefit from huge pages. Their approach is to recover from fragmentation rather than preventing the fragmentation beforehand, thus inducing management overheads.

All the above approaches either create separate regions for distinguishing various types of allocations or try to recover from fragmentation once higher order pages are not available. None of the methods mentioned above puts the extra effort to decide on an action plan when enough memory is not available in kernel region and it has to fallback to user region for kernel memory. Even with our minimal modification of kernel and simple design shows significant improvements with respect to huge page allocations.

Another orthogonal approach which one can follow is through minimizing the kernel memory footprint itself. Aravinda et al. came up with Prudence, a dynamic memory allocator that is tightly integrated with the synchronization mechanism to ensure visibility of deferred objects to the memory allocator. Such an integration enables Prudence to (i) identify the safe time to reclaim deferred objects memory, (ii) have an inclusive view of the allocated, free and about-to-be-freed objects, and (iii) exploit optimizations based on the hints about the future during important state transitions.

While all the above methods do not actually tackle the core problem of non movable kernel

pages, some have tried to make parts of kernel itself movable. This patch enables moving of balloon driver pages which are largely used in virtualization. There is a fair amount of work required to support compaction in an arbitrary kernel subsystem. As a result, this support is likely to be confined to a relatively small number of subsystems that use substantial amounts of memory. Gioh's patch adapts the balloon driver subsystem in this way; on systems employing virtualization, balloon devices can (by their nature) use large amounts of memory, so making it movable makes some sense. Other possible use cases include long-lived I/O buffers or drivers (such as graphics drivers) that need to store large amounts of data.

Chapter 4

DESIGN AND IMPLEMENTATION

In this chapter, we discuss four approaches to tackle fragmentation due to kernel page placements. Each has its own pros and cons with respect to running time efficiency, success rate and novelty.

4.1 Optimal Page Block Selection (OPBS)

We define an optimal page block as the one that would result in *maximum* pages getting reserved during kernel memory fallbacks. In OPBS, all order lists are linearly scanned to search the best possible page block which has maximum number of free pages to reserve for future kernel requests. To facilitate quick look-up for the number of free pages in a page block we maintain free page count in each page block data structure. This allows us to check for free page count in constant time.

Following are the issues related to OPBS

- **Time consuming page block scanning:** System having very large memory broken in smaller pages might lead to very long scanning process of page blocks. This may significantly increase the page allocation time.
- **Running time inefficiency:** Running time for this technique is linear, i.e. it will scan all page blocks.

Even though the cost of scanning page blocks is high, following optimizations make fallback a rarely executed code path in practice:

- The kernel is a highly organized system and its memory footprint is very low compared to user applications. Hence, its contribution is a tiny fraction of all memory requests in the system.

- Slab allocators [4] have been implemented on top of buddy allocator to serve memory requests of kernel objects. It is only when slab pages get exhausted that kernel memory request is handled by buddy allocator.
- Memory domains tend to balance their size depending on the workloads. Hence, majority of kernel allocations come from kernel domain.

4.2 Active Anti Fragmentation (AAF)

Active anti-fragmentation [12], originally proposed by Joonsoo Kim, has been discussed in the community as another potential optimization for improving upon fragmentation in Linux. The idea behind AAF is to always reserve an entire page block during kernel memory fallbacks. In this approach when the kernel request fallbacks to a user page block, all user pages already allocated in that page block are migrated to a different page block so as to reserve this free pages for future kernel requests.

AAF is the closest approach to page clustering in Linux, as it strictly tries to reserve all available pages for kernel even when they are allocated. It not only reserves the free pages but moves all allocated user pages elsewhere in memory for future kernel requests.

Following are the issues related to AAF

- **Migrating pages overhead:** AAF will move all the movable pages elsewhere in memory to reserve maximum pages for future kernel requests. This is costly in terms of allocation of pages.
- **Lock Contention:** AAF technique uses locks to move pages and it may incur some delay.
- **TLB shootdowns:** We observed an order of magnitude more TLB shootdowns with a kernel memory intensive workload like kernel build.

4.3 Incremental Page Block Selection (IPBS)

Incremental page block selection adapts to the fragmentation in system and makes a clever page block choice to avoid the spread of kernel pages. IPBS improves on the running time efficiency of OPBS by trading slight optimality. It essentially works at 4 levels, which are described as follows.

LOW: System boots and remains in LOW till page block size pages are available for reservation on kernel fallbacks. Here the behavior of the allocator is similar to the one on default

Algorithm 2 *

Algorithm-2: Incremental Page Block Selection (IPBS)

```
1: fallback_kernel_IPBS(request_order)
2: ....
3: if level = LOW then
4:     page = get_head_page(USER, request_order)
5: else if level = MEDIUM then
6:     page = get_random_page(USER, request_order)
7: else if level = HIGH then
8:     page = get_optimal_page_in_list(USER, request_order)
9: else /* CRITICAL */
10:    page = get_page_OPBS(USER, request_order)
11: end if
12: ....
13: adjust_level();
14: return page
```

kernel which facilitates speedy allocation. Here the overhead involved for kernel fallback is non-existent. As the allocator is able to allocate page block size pages on kernel fallbacks, allocator infers that fragmentation is not a concern as of now and hence, it allocates in default manner.

MEDIUM: Once the allocator fails to allocate page block size pages for kernel fallback a situation we call MEDIUM fragmentation, it switches to MEDIUM in which it randomly picks 2 page blocks and allocates the one having more number of free pages which leads to more pages being reserved for future kernel requests. This ultimately leads to fewer fallbacks for kernel.

HIGH: The allocator switches to HIGH once the pages being allocated for kernel fallback are of order lower than 7. HIGH will scan only requested order list up to 64 page blocks to find the most suitable candidate, this ensures that we remain proactive in delaying the fragmentation.

CRITICAL: Once the allocator realizes that the system is under intense fragmentation it switches to OPBS with limit of 64 page block scan in each list and hopes to delay compaction as much as possible. We call this modified OPBS with 64 page block limit as Restricted Page Block Selection (RPBS).

The strategy works effectively and keeps the fragmentation at low and at the same time is efficient. The key idea is to monitor the kernel fallback page size and make judicious choice from the available page block candidates. This ensures that it adapts to the fragmentation caused by kernel page placements and take proactive measures to negate it.

Following is the issue related to IPBS

- **Running time inefficiency:** Once IPBS disintegrates to RPBS the running time of the algorithm can still be significant.

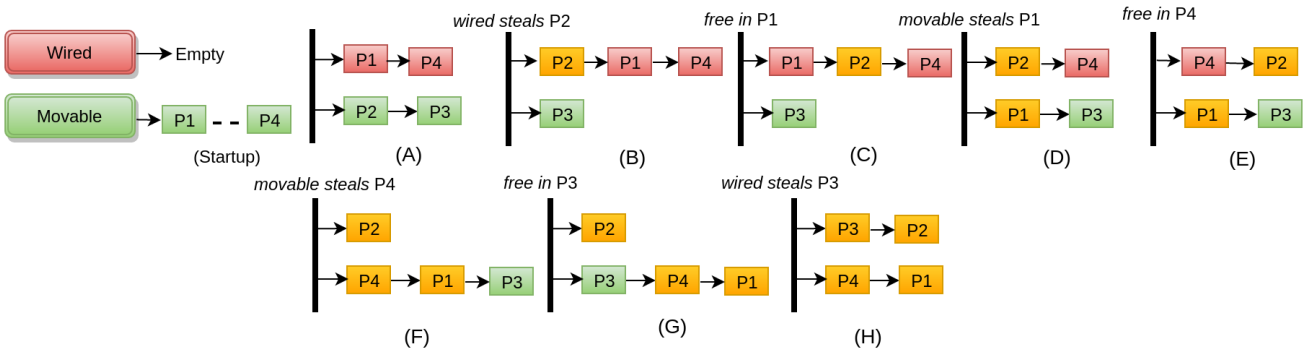

Figure 4.1: Impact of stealing on huge pages and pageblock pollution. Red and green blocks represent huge pages containing only wired and movable pages respectively while yellow represents pageblocks containing both movable and wired pages.

4.4 Illuminator

In this section, we explore the inefficiency of page-clustering in mitigating the impact of kernel memory (we call kernel memory as *wired* memory and user memory is referred as *movable* memory in this section) on huge pages which was the motivation for our final solution.

4.4.1 Anti-fragmentation and composite huge pages

Page-clustering labels each huge page with a color, let us say red (for wired zone) and green (for movable zone). A 2-coloring approach, however, works well only as long as wired and movable memory can be placed on isolated huge pages. Once memory pressure increases, zones start to steal pages from each other. Since pages are always allocated from and freed to head of free lists, stealing can quickly pollute all huge pages with kernel memory. This adverse behavior of stealing based page-clustering is best descried with an example.

Let us say a system starts with 4 huge pages i.e., P1-P4, all of which belong to movable zone at startup as shown in Figure 4.1. Page-clustering works well until system reaches state A where P1, P4 belong to red and P2, P3 belong to green zone. At this point, P2 will be stolen if P1 and P4 have no free page and kernel attempts an allocation. It would be added to wired zone and contain both wired and movable pages as system reaches state B. We call such a huge page as *composite huge page* and represent it with yellow color hereafter. Now consider this sequence: a free operation in P1 followed by movable zone stealing it. This sequence would convert P1 into a composite huge page as shown in Figure 4.1 through system transition from state B→C and C→D. Similar sequence of free and stealing operations would convert first P4 and then P3 into composite huge pages via state transitions D→E, E→F, F→G and G→H. At

this point, all huge page allocations will fail unless some wired memory is dropped explicitly by the kernel. It is important to note that it took just 7 allocation/free operations for system to reach state H (where all huge pages contain at least one wired page) from state A.

It is possible that sufficient free memory becomes available in the system with no significant reduction in wired memory. Such a scenario represents a lost opportunity where huge pages could have been helpful.

We observed that the behavior of page-clustering can be optimized as: while stealing, steal from an already polluted page block if possible. For example, consider system transition from state E to state F, where P1 was already polluted but P4 was stolen by wired zone. At this point, stealing P1 (if it has some free memory) would protect P4 from getting polluted. Reaching till P1, however, will involve list traversal which can be of arbitrary length and expensive.

4.4.2 Composite huge pages and memory compaction

Composite huge pages are not visible to memory compaction which is detrimental to system performance in two ways. First, pages are migrated unnecessarily when a green composite huge page falls in the path of migrate scanner. Second, it induces uncertainty in the behavior of memory compaction as the location of composite huge pages determine the success of memory compaction.

We present *Illuminator*, a framework to manage composite huge pages that shows memory allocators what they need to see.

4.4.3 Design and Implementation

As show in Figure 4.2, the buddy allocator now classifies composite huge pages separately. When wired zone is unable to satisfy the request for wired memory it first checks in movable zone if it is able to steal completely empty huge page. On success it will use that stolen huge page for future wired memory requests. Else it steals from composite zone which already contains huge pages which are polluted by wired pages. Similarly, when movable zone is unable to satisfy user memory request it first checks if can steal a completely empty huge page from wired zone and if it succeeds in doing so it reserves that huge page. Else it steals pages from composite zone.

Illuminator also tries to recover pages from composite zone to movable zone as and when the wired pages get freed from a composite huge page. Whenever there are no wired pages present in composite huge page the composite huge page is moved to movable zone so that user applications can make use of that huge page. The allocation and freeing are both happening in constant time and thus Illuminator has zero overhead with respect to allocation and freeing.

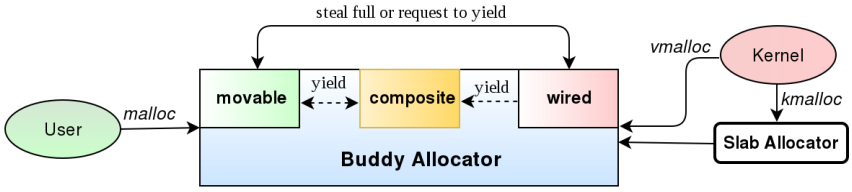

Figure 4.2: Wired zone and movable zone steal only huge page or larger size pages from each other if available, else they steal from composite.

We consider the same example for Illuminator that we explained for default buddy allocator earlier. As shown in Figure 4.3, the system again starts with 4 huge pages i.e., P1-P4, all in movable zone. Till the system reaches state A, the page clustering works well and the huge pages are distributed to respective zones according to the need. At this point of time, kernel attempts an allocation and it fails because of insufficient base pages in P1 and P4. This then leads to stealing attempt from movable, but as movable does not completely free huge page the attempt fails. Then the wired zone tries to allocate from composite, in which case composite first steals from movable and then satisfies the wired request. Note that now the huge page P2 contains both movable and wired pages which is represented by yellow color in the diagram. This huge page is categorized as composite due to the presence of both wired and movable pages. Later on at point B, movable receives a request for allocation and it fails due to insufficient free pages in P3. And so it tries to steal a completely empty huge page from wired zone. This attempt fails as kernel (wired) does not have such huge page. Then the request is made to composite for the allocation which also fails due to insufficient free pages in P2, which leads to stealing from wired, i.e free pages from huge page P1. After this P1 now is categorized as composite due to presence of both wired and movable pages. Then at later point of time when some base page free happens in huge page P2 and wired pages are no more present in P2, the huge page is recovered from composite and put into movable for future user applications that might need huge page. This situation is represented by X in the figure. Here we see that the situation that happened earlier with default buddy allocator does not repeat with Illuminator and eventually we recover some huge pages back to movable when all wired pages are freed from composite huge page.

By making compaction aware of the composite huge pages we reduce the overhead of compaction by skipping the whole composite huge page when compaction is called to accumulate huge page size region. The migrate scanner of the compaction need not scan the inner base pages of the composite huge page as migrating the inner base will not help in accumulating a huge page size region.

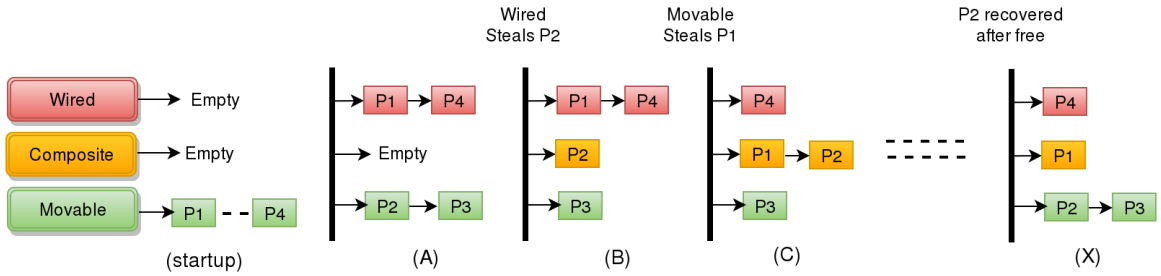

Figure 4.3: Stealing in Illuminator, where we limit the the number of composite huge pages and also recover from composite.

4.5 Kernel Modifications

The final implementation of IPBS comes to roughly 600 lines of code including insertion, deletion and modifications in Linux-4.1.13. Illuminator was implemented on Linux-4.5 and the modifications were around 550 lines of code. Major part of these changes is contributed by the page allocation and page freeing operations. Other changes represent the introduction of page block data structure which holds the number of free pages in the page block and number of kernel pages allocated from page block. Several functions are introduced to select optimal page block and also modifications related to a new region called mixed region. The code changes are minor which can be easily integrated into mainstream kernel tree.

4.6 Running time complexity

The running time efficiency of IPBS may degenerate to OPBS which is $O(n)$, where n is the number of page blocks that contain a free page which can be used for current request. We use RPBS (OPBS with 64 page block limit while scanning) when fragmentation is critical which scans all order lists for suitable page selection. The running time efficiency of OPBS is $O(n)$, IPBS is $O(n)$ due to the above mentioned reason while that of AAF is $O(1)$ + cost of moving pages. Running time complexity of Illuminator is $O(1)$.

Table 4.1: Run time complexity

Kernel	**Running time complexity**
Default	$O(1)$
AAF	$O(1)$ + cost of moving pages
OPBS	$O(n)$
IPBS	$O(n)$
Illuminator	$O(1)$

Chapter 5

RESULTS AND EVALUATION

5.1 Experimental Setup

Our primary setup is a server machine equipped with Intel® processor with 4*2 cores and LLC (Last Level Cache) size of 8MB. Size of physical memory varies between 2GB and 4GB depending on the benchmarks. Some of the benchmarks like Gorman [8] recommend to use 3GB memory and maximum up to 4GB. Fragmenting a system can take days in larger memory setting and variations can be significant. Therefore, we run our benchmarks on 4GB or less memory configuration. *cpuid* reports d-TLB supporting 64 entries for 4KB and 32 entries for 2MB pages while i-TLB contains 64 entries for 4KB and 8 entries for 2MB pages. The system provides robust support through hardware counters for measuring system wide energy consumption across CPU cores, package (cores + LLC) and DRAM which can be accessed through RAPL interface [18] available in *perf*.

5.1.1 Unusable Free Space Index

Table-5.1 depicts the number of tainted page blocks as well as the *unusable free space index (UFSI)* for huge pages (i.e., order 9) on our system with 4GB physical memory. We run kernel

Kernel	# Tainted page blocks	UFSI
Default	1683	0.61
AAF	134	0.24
OPBS	508	0.21
IPBS	874	0.31
Illuminator	336	0.28

Table 5.1: Unusable free space index for huge pages (order 9) and huge page success rate with different kernels on 4GB system.

Kernel	HP Success Rate	Slowdown (%)
Default	0%	0
AAF	5%	8
OPBS	8%	3
IPBS	7%	1
Illuminator	16%	0

Table 5.2: Number of tainted page blocks and slowdown with different kernels.

builds for measuring the number of tainted kernel page blocks. Once the kernel builds are done we record the number of tainted page blocks. Note that though AAF performed better in terms of restricting kernel memory within fewer page blocks, it is OPBS that wins for *unusable free space index*. Further analysis revealed that it was indeed happening due to the conflict between memory compaction and anti-fragmentation as AAF page block selection was aligned towards KML configuration discussed earlier in chapter 2. It also confirms our earlier observation that minimizing tainted page block count is not sufficient for controlling fragmentation. IPBS performs reasonably well in term of both restricting fallbacks to few page blocks as well as reducing the *unusable free space index*. Illuminator outperforms all if we consider the combined results.

5.2 Results

5.2.1 Reduction in Compaction Efforts

Illuminator reduces the compaction effort significantly by avoiding unnecessary migration of pages. We measure the compaction efforts by running 2 sequential kernel builds which severely fragments memory. Once the system is fragmented, we compact memory using *proc* interface which invokes full compaction.

Figure-5.1 shows us that with increase in the Transparent Huge Page success rate we also see reduction in allocation time and energy consumed per huge page allocation. In Illuminator, we either succeed or fail quickly in terms of huge page allocation. That results in substantial gain, which comes through skipping unnecessary scanning and migration of pages.

5.2.2 Benchmarking Huge Page Allocations

In an another experiment, we use *stress-highalloc* from *mmtests* [8] for benchmarking the success rate of huge page allocations under rigorous memory pressure on a 3GB system. It attempts to allocate huge pages three times after severely fragmenting physical memory with kernel compilation. In the first attempt, when the success rate of default kernel was 0%, IPBS was

	Huge pages			Time (ms per huge page)			Energy (µJ per huge page)		
	Linux	**Illuminator**	**Gain**	**Linux**	**Illuminator**	**Gain**	**Linux**	**Illuminator**	**Gain**
Min	59	386	6.5x	3.42	0.89	3.84x	35.1	8.7	4.03x
Max	118	460	3.9x	4.70	1.02	4.60 x	46.0	10.0	4.60x
Avg	89	421	4.7x	4.06	0.92	4.41x	40.8	9.1	4.48x

Figure 5.1: Effort spent in accumulating huge pages by Illuminator.

able to allocate 7% of system memory as huge pages. When system was at rest in the last attempt, IPBS was able to satisfy 37% of huge page allocations as compared to 13% of default kernel. As shown in Table-5.2 we measured the success rate of different kernels out of which the best success rate was achieved by Illuminator.

We also measured the huge page allocation success rate on 4GB systems and the results were impressive. While default kernel was able to allocate 39% of huge pages, Illuminator was able to allocate 78% of huge pages. This difference of 39% in huge page allocation turns out to be around 800 more huge pages with Illuminator. Additional success rate of IPBS was found to be varying between 25% and 30% with 4GB physical memory as well.

5.2.3 Performance/Energy Measurement

39% additional success rate yields more than 800 huge pages on a 4GB system. We benchmark the benefits of these pages, with both using *libhugetlbfs* library support and THP support available for Linux systems, with some real applications. 800 huge pages were reserved before running the test applications which include *stream*, *canneal* (from PARSEC benchmark suite), a few *SPECjvm2008* benchmarks and some *SpecCPU2006* benchmarks. We show the improvements with THP on Virtual Machines, SpecCPU2006 benchmarks as well.

We measure the impact of additional huge pages on system TLB with the above mentioned benchmarks. Reduced TLB load ultimately results in improved performance and energy savings as system spends less time in translating virtual to physical addresses.

The runtime and throughput gains are shown in Table-5.4. Stream reports upto 20% reduction in runtime with default input array size while canneal benefits upto 13% with native input set. Throughput gain for SPECjvm applications is also substantial (upto 6%). Also note that huge page benefit varies across platforms depending on the system configuration (e.g.,

TLB and Cache size). On an Ivy-Bridge cluster server, we observed 50% improvement in runtime for stream and upto 12% throughput gain with SPECjvm applications which implies that fragmentation management could be more crucial in server space.

Figure-5.3 shows energy savings across cores, LLC and DRAM for our test applications. Our analysis shows energy gains to be high as compared to performance which further adds to the motivation for better fragmentation management. For example, performance gain of 20% in case of stream translates into 27% energy reduction. Note that energy savings shown here are not absolute gains from the overall system perspective and the cost of page migration incurred to facilitate the allocation of huge pages need to be taken into account. Further analysis shows that it costs 5 Joules of energy to migrate 400MB pages across cores, LLC and DRAM which is quite low and can be easily amortized with savings achieved with huge pages.

5.3 Performance Evaluation

5.3.1 SPECjvm2008

SPECjvm2008 (Java Virtual Machine Benchmark) is a benchmark suite for measuring the performance of a Java Runtime Environment (JRE), containing several real life applications and benchmarks focusing on core java functionality. The suite focuses on the performance of the JRE executing a single application; it reflects the performance of the hardware processor and memory subsystem, but has low dependence on file I/O and includes no network I/O across machines. The SPECjvm2008 workload mimics a variety of common general purpose application computations. These characteristics reflect the intent that this benchmark will be applicable to measuring basic Java performance on a wide variety of both client and server systems.

SPEC also finds user experience of Java important, and the suite therefore includes startup benchmarks and has a required run category called base, which must be run without any tuning of the JVM to improve the out of the box performance.

Benchmarks like compress, crypto.rsa, derby and sunflow showed throughput improvements due to the use of huge pages. Improvements in Figure-5.2 comes from the fact that the translation cost is now reduced which reduces the stressing of TLB. As this benchmarks run for a fixed duration energy savings are not relevant here.

5.3.2 Stream

Stream checks for the following type of operations:

- **Copy:** measures transfer rates in the absence of arithmetic.

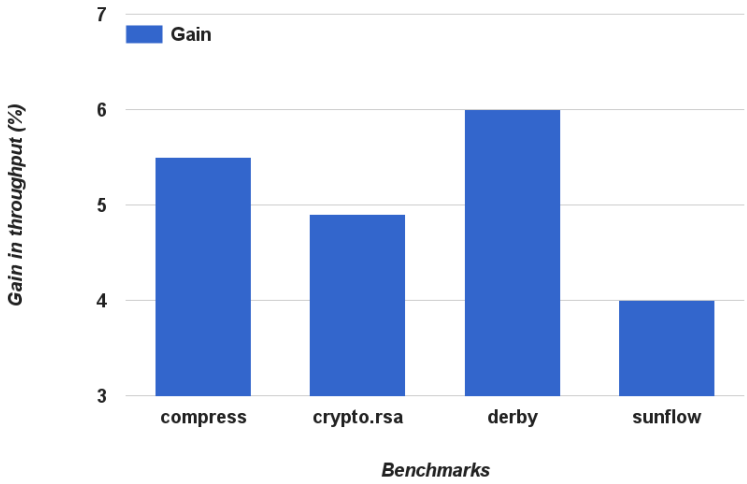

Figure 5.2: Gain in throughput for few of the SpecJVM benchmarks

- **Scale:** adds a simple arithmetic operation.
- **Sum:** adds a third operand to allow multiple load/store ports on vector machines to be tested.
- **Triad:** allows chained/overlapped/fused multiply/add operations.

We see significant gain in terms of runtime and energy performance and that is largely due to the reduced TLB misses as shown in table-5.3. Reduction in TLB misses for stream was around 72% which goes on to suggest that huge pages are critical to such applications.

5.3.3 Canneal from Parsec

This kernel was developed by Princeton University. It uses cache-aware simulated annealing (SA) to minimize the routing cost of a chip design. SA is a common method to approximate

Benchmark	**TLB Miss Ratio**		
	W/O HP	**W HP**	**Reduction**
Canneal	6.46%	0.18%	97%
Stream	0.07%	0.02%	72%

Table 5.3: TLB miss ratio for canneal and stream with (W HP) and without huge pages (W/O HP).

Benchmark	Runtime (milliseconds)		
	W/O HP	W HP	Gain
Stream	1549	1237	20%
Canneal	90360	78940	13%

Table 5.4: Runtime performance gain on canneal and stream with (W HP) and without huge pages (W/O HP).

the global optimum in a large search space. Canneal pseudo-randomly picks pairs of elements and tries to swap them. To increase data reuse, the algorithm discards only one element during each iteration which effectively reduces cache capacity misses. The SA method accepts swaps which increase the routing cost with a certain probability to make an escape from local optima possible. This probability continuously decreases during runtime to allow the design to converge. The program was included in the PARSEC program selection to represent engineering workloads, for the fine-grained parallelism with its lock-free synchronization techniques and due to its pseudo-random worst-case memory access pattern.

Canneal benefits from huge pages in a large way since the TLB miss rate comes down by 97% which improves the running time by 13%. The reduced running time give us energy gain of around 14%.

5.3.4 SpecCPU2006

SPEC CPU2006 is a suite of benchmark applications designed to test the CPU performance. The suite is composed of two sets of tests. The first being CINT (aka SPECint) which is for evaluating the CPU performance in integer operations. The second set is CFP (aka SPECfp) which is for evaluating the CPU floating point operations performance.

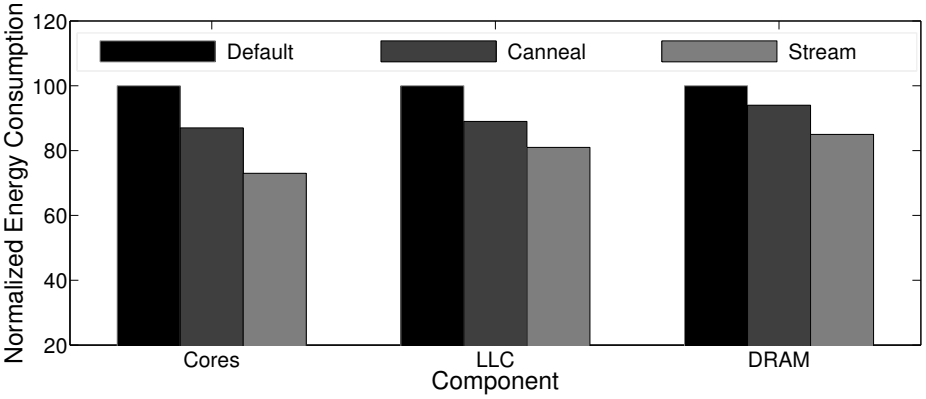

Figure 5.3: Normalized energy consumption of stream and canneal with additional huge pages.

Benchmark Name	TLB miss reduction (%)	Execution time gain	Energy gain
429.mcf	88%	39%	38%
459.GemsFDTD	83%	14%	12%
471.omnetpp	68%	13%	12%
473.astar	70%	9%	8%
436.cactusADM	50%	20%	19%
410.bwaves	76%	4%	3%

Table 5.5: TLB miss reduction, execution time and energy gain for some of the SpecCPU2006 programs.

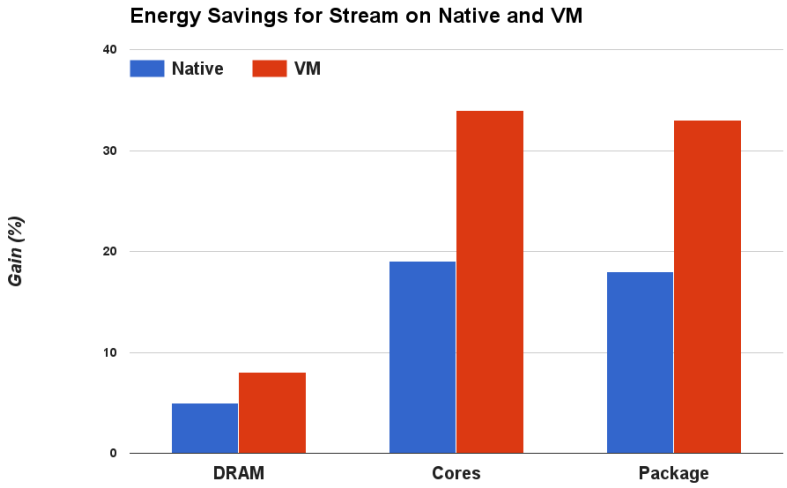

Figure 5.4: Energy savings for native hardware and virtual machine with huge pages.

The benchmark applications are programs that perform a strict set of operation that simulate real time situations, such as physical simulations, 3D graphics, and image processing. These applications are written in different programming languages, C, C++ and Fortran. Many SPECfp benchmark applications are derived from applications that are freely available to the public and each application is assigned a weight based on its importance.

Quite a few benchmarks from SpecCpu2006 get advantage of huge pages. Benchmarks like 429.mcf and 436.cactusADM show significant improvement in terms of running time and energy improvements.

5.3.5 Virtual Machines

We also ran experiments to check if virtual machines benefit from huge pages as the TLB miss cost in virtual machine(VM) environment can be quite significant as compared to native machine. Most Virtual Machine Managers (VMM) use a shadow page tables or nested page tables to limit the two level translation cost. But when a TLB miss is incurred it may cost 16 memory access in case of nested page table. The fact that there is 2 level translations happening in case of virtualization i.e., 1st in guest OS and the other in host OS is the reason behind better results in case of virtualization.

We use VirtualBox from Oracle as the VMM and run Fedora 23 as both host and guest operating systems to measure the impact of huge pages on VM's. We enable THP in the guest OS as well in the Host OS before running canneal and stream benchmarks. As seen in Figure 5.4 the runtime and energy improvements are substantial which reaffirms our view that huge pages will benefit VM environment.

5.3.6 Biobench

We select mummer, a genome level alignment application, and tigr, a sequence assembly application, from biobench benchmark suite. These applications mostly operate on strings which is quite different from integer and floating point applications of Spec CPU2006.

5.4 Huge page allocation

We show the success rate of huge page allocation with real application on default Linux and Illuminator. For this experiment we run kernel compilation twice which puts the systems in fragmented state. Then we run this applications sequentially and measure the successful huge pages allocated to each applications during their run time. Figure-5.5 shows the gain in terms of huge page allocation for various real applications from Biobench, Spec CPU2006 and Parsec. Illuminator was able to allocate significantly more huge pages due to its composite huge page framework as well as better coordination with compaction. This implies that with higher allocation success rate of huge pages in Illuminator, it is able to out perform default Linux in terms of execution time and energy consumption.

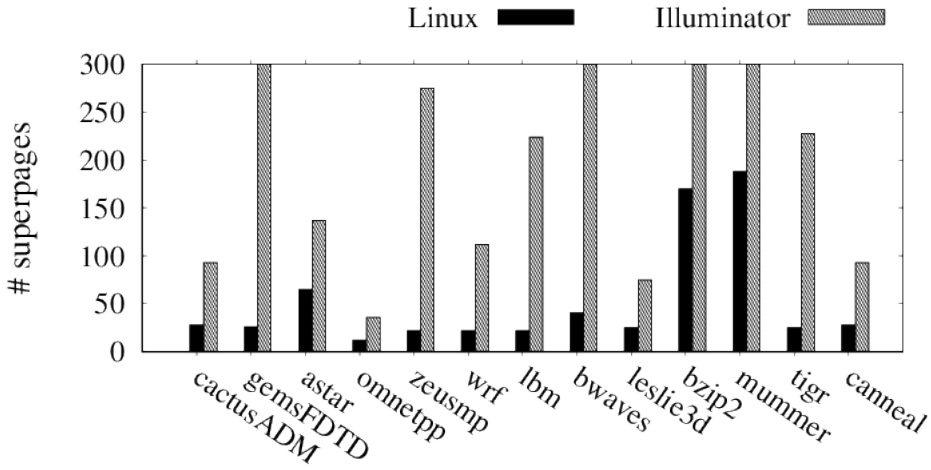

Figure 5.5: Number of huge pages (superpages) successfully allocated to each application.

Chapter 6

CONCLUSIONS

In this thesis, we discussed the interaction of kernel page placement with memory compaction and anti-fragmentation framework. We proposed 2 simple allocators namely, OPBS and IPBS which with minimum effort gives better performance in terms of reducing fragmentation. We introduced a new domain with mixed page blocks using which we proposed a new page allocator which takes care of controlling fragmentation at huge page granularity. We show how fragmentation which occurs due to kernel page placement has severe effects on the huge page allocation success rate and how huge pages could have benefited the application in terms of running time at the same time being energy efficient. We also evaluated the benefits of these huge pages with real applications.

6.1 Future Work

As we have shown memory compaction and anti-fragmentation policy of Linux does not coordinate with each other and compaction can worsen the fragmentation in the system. This calls for the a tightly coupled memory compaction and anti-fragmentation framework which can potentially reduce this fragmentation. Other direction is to reduce the kernel memory footprint itself by introducing coordination between buddy allocator with slab allocator. The ultimate goal is to make kernel pages movable so as to remove the core issue of non-movable pages which thwarts the success of compaction.

References

[1] Specjvm2008, 2008. URL https://www.spec.org/jvm2008/. https://www.spec.org/jvm2008/.

[2] V. Babka. Fighting physical memory fragmentation with memory compaction, 2014. URL http://labs.suse.cz/vbabka/compaction.pdf. http://labs.suse.cz/vbabka/compaction.pdf.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] J. Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994. 19

[5] J. Corbet. Slab defragmentation, 2007. URL https://lwn.net/Articles/236108/. https://lwn.net/Articles/236108/.

[6] J. Corbet. Memory compaction, 2010. URL https://lwn.net/Articles/368869/. https://lwn.net/Articles/368869/.

[7] J. Corbet. Making kernel pages movable, 2015. URL https://lwn.net/Articles/650917/. https://lwn.net/Articles/650917/.

[8] M. Gorman. Mmtests: Benchmarking framework primarily aimed at linux kernel testing. URL https://github.com/gormanm/mmtests. https://github.com/gormanm/mmtests. 25, 26

[9] M. Gorman and P. Healy. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management*, pages 41–50. ACM, 2008. 16

[10] M. Gorman and A. Whitcroft. The what, the why and the where to of anti-fragmentation. In *Proceedings of the 2006 Ottawa Linux Symposium*, OLS '06, pages 369–384. 3, 6, 7, 12, 15

[11] M. Gorman and A. Whitcroft. Supporting the allocation of large contiguous regions of memory. In *Linux Symposium*, page 141, 2007. 8, 15

[12] J. Kim. mm: support active anti-fragmentation algorithm, 2015. URL https://lkml.org/lkml/2015/4/27/94. https://lkml.org/lkml/2015/4/27/94. 19

[13] S.-H. Kim, S. Kwon, J.-S. Kim, and J. 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. ISBN 978-1-4503-3589-8. URL http://doi.acm.org/10.1145/2754169.2754179. 16

[14] J. Mauro and R. McDougall. *Solaris Internals (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006. ISBN 0131482092. 2

[15] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[16] M. K. McKusick and G. V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004. ISBN 0201702452.

[17] J. Navarro, S. Iyer, P. Druschel, and A. 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*, 2002. URL http://www.usenix.org/events/osdi02/tech/navarro.html. 16

[18] J. Pan. Rapl (running average power limit) driver, 2013. URL https://lwn.net/Articles/545745/. https://lwn.net/Articles/545745/. 25

[19] A. Panwar and K. Gopinath. Towards practical page placement for a green memory manager. In *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*, pages 155–164, 2015. URL http://dx.doi.org/10.1109/HiPC.2015.42.

[20] A. 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. ISBN 978-1-4503-4091-5. URL http://doi.acm.org/10.1145/2872362.2872405.

[21] R. Shamsudeen. Performance tuning: Hugepages in linux, 2008. URL https://www.pythian.com/blog/performance-tuning-hugepages-in-linux/. https://www.pythian.com/blog/performance-tuning-hugepages-in-linux/.