A Case for Protecting Huge Pages from the Kernel

Ashish Panwar [*]
Intel Corporation &
Indian Institute of Science
ashish.panwar@csa.iisc.ernet.in

Naman Patel [*]
Indian Institute of Science
patel.naman@csa.iisc.ernet.in

K. Gopinath
Indian Institute of Science
gopi@csa.iisc.ernet.in

Abstract

Controlling memory fragmentation is critical for leveraging the benefits of huge page support offered by modern architectures. The division of free memory into non-contiguous regions over time restricts huge page allocations in long run. Compaction is a popular mechanism to recover contiguous blocks of free memory on demand. However, its success rate of accumulating free regions at huge page granularity (e.g., 2MB in x86_64) is low in most situations due to the presence of unmovable kernel memory. Hence, a prudent page placement algorithm is required to control fragmentation and protect huge pages against kernel memory allocations, in order to sustain system performance over long periods of time.

In this work, we explore the interaction of kernel pages with fragmentation avoidance and recovery mechanism in Linux. Our analysis shows that stock kernel memory layout thwarts the progress of memory compaction. Furthermore, compaction can potentially induce more fragmentation depending on where kernel memory was placed by the underlying page allocator. We discuss the scope of optimization in current Linux framework and show how an effective fragmentation management can yield up to 20% performance improvement and up to 27% energy savings with the help of additional huge pages.

Keywords Huge pages, Buddy allocator, Anti-fragmentation, Memory compaction

1. Introduction

Complex and large working sets of modern applications are known to gain substantial benefits with huge pages [18]. 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 superpages 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 [3]. Huge pages can minimize this translation cost by mapping large portions of process address space into a single TLB entry. However, facilitating allocation of large contiguous blocks is a challenge for operating systems because of fragmentation [14].

Fragmentation, both internal and external, occurs 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 [12]. Internal fragmentation occurs when a larger block than the request size is allocated. Both results in inefficient utilization of resources and performance/energy loss in many cases. In the context of this paper, 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 long running systems [2]. 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 [7]. Operating systems typically have a large pool of non-pageable kernel memory objects [15, 17] 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.

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 contigu-

[*]The authors contributed equally for this work.

APSys '16, August 04-05, 2016, Hong Kong, Hong Kong
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4265-0/16/08...$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2967360.2967371

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:      $nr\_reserved = reserve\_pblock\_pages(page)$
10:     if $nr\_reserved >= pblock\_nr\_pages/2$ then
11:         $change\_pblock\_domain(page,\ KERNEL)$
12:     end if
13:     $/ * split\ if\ page\ is\ large,\ before\ returning\ * /$
14:     return $page$
15: else
16:     return $NULL$
17: end if
```

ous blocks (that constitute huge pages) from kernel memory allocations. Linux utilizes *page clustering* based anti-fragmentation [11] 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 impacts memory subsystem and their interference with compaction can actually exacerbate fragmentation in long running systems.

Our primary contributions in this paper are:

1. Evaluating the impact of different kernel page layouts on memory fragmentation and compaction.
2. Exploring the scope of optimization in a recent Linux kernel and benchmarking performance/energy gain of a refined anti-fragmentation framework with real applications.

2. Background and Motivation

Linux manages free memory using a variant of binary buddy allocator in 11 lists i.e., order 0 to 10 where each order list links together power of 2 contiguous pages. Furthermore, a *pageblock* denotes an order 9 page i.e., a contiguous block of 512 pages. In the rest of the paper, we refer to pageblock as *pBlock*, kernel pages as *sysPages* and application pages as *usrPages*. Fragmentation is dealt with a combination of avoidance and recovery mechanisms discussed in following subsections.

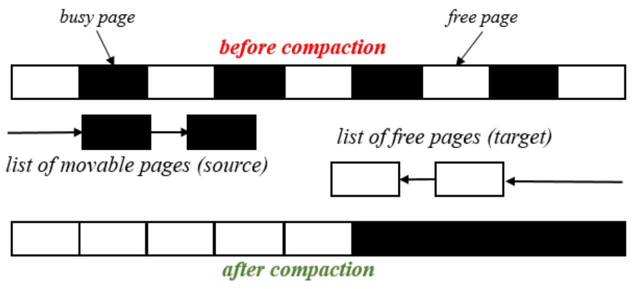

Figure 1: Physical memory before and after compaction.

2.1 Anti-Fragmentation

Linux implements page clustering [11] to avoid fragmentation by partitioning physical memory in different regions. For simplicity, we broadly classify these regions in two domains i.e., kernel and user which are intended to serve system and application memory demands respectively. Each domain contains a group of pBlocks[1]. Note that the size of a pBlock is aligned to the default huge page size of x86_64 i.e., 2MB. More memory can be potentially used to satisfy huge page allocations if kernel space is restricted to a few pBlocks. Page clustering attempts to prevent kernel from occupying large pool of huge pages by placing its memory in a few pBlocks.

2.2 Memory Compaction

Compaction tries to accumulate small non-contiguous free regions into larger contiguous regions by migrating pages in memory. For the purpose of understanding, compaction as shown in can be thought of as a combination of two scanners:

1. The first one starts at the bottom of memory and prepares a list of in-use usrPages. We refer to them as source pages hereafter.
2. The other collects free pages from the top of memory which we call as target pages.

Once the two scanners meet, source pages are copied onto target list and their page table references are updated properly (see Figure 1).

2.3 Motivation

The real problem with respect to fragmentation appears when kernel memory starts growing beyond its domain. When this happens, sysPages need to be allocated from user domain (we call this event as memory fallback). During fallback, the first page[2] from the highest order non-empty buddy list is selected for allocation and free pages from its pBlock are stolen for future kernel memory requests. Algorithm-1

[1]Note that a pBlock is a contiguous region of physical pages. However, different pBlocks in a domain need not be contiguous.

[2]In long running systems, free lists get randomized and arbitrary pages are found at the head of free lists. We call the first page as random in the rest of the paper.

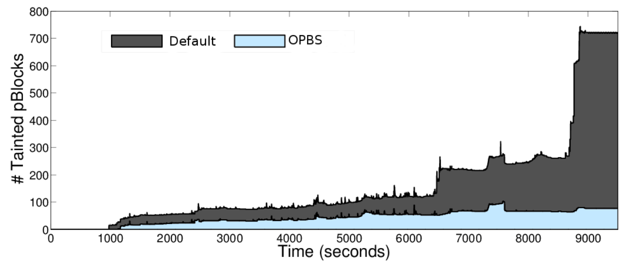

Figure 2: Tainted pBlock formation rate with default and OPBS (explained in Section 4.2) algorithms on 2GB physical memory.

shows the code snippet for kernel memory fallback. However, a random page selection results in an inefficient realization of page clustering causing interleaving of sysPages and usrPages on majority of pBlocks in long run. Compaction also suffers due to this interleaving as pBlocks containing sysPages can not be freed entirely. Over a period of time, the system finds it difficult to provide applications with huge pages. In the rest of the paper, we refer to pBlocks containing both usrPages and sysPages as *tainted* pBlocks.

Figure 2 shows the rate of tainted pBlock formation with default kernel fallback routine as compared to a modified algorithm (referred to as OPBS in the figure which is our optimal version). The modified algorithm is designed to select a pBlock that can reserve maximum pages during fallbacks. Our analysis shows that there is sufficient scope for optimizing fallbacks for controlling the layout of kernel pages. Moreover, it is a rarely executed code path and hence such optimizations can be achieved without sacrificing the overall performance of memory allocator.

3. Defining the Problem Experimentally

Many real-world applications which use large amount of kernel memory like database applications [22] and kernel compilation induce fragmentation. We run kernel builds for analyzing Linux fragmentation management framework on a 4GB system. A detailed analysis of our experiments is discussed below:

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

Table 1: Distribution of sysPages across all user pBlocks. Note that more than 90% pBlocks contain less than 10% sysPages.

3.1 Issues with Anti-Fragmentation

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

1. A random pBlock selection during fallback 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 pBlocks.
2. When fallbacks reserve pages without updating the pBlock ownership (i.e., *if* condition fails on line 10 in Algorithm-1), the pBlock gets divided between two domains. While its free pages are allocated as kernel memory, pages that were already allocated from it are freed back to the user domain.

Such behavior can cause a large number of pBlocks containing sysPages in long run. Note that none of the pBlocks containing sysPages 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 pBlocks. Further, we define for each user pBlock the degree of pollution as the fraction of its memory allocated as sysPages. Table 1 shows the degree of pollution for all user tainted pBlocks at the end of two successive kernel builds with 4GB physical memory. We find more than 80% (1345 out of 1667) user pBlocks being tainted while majority of them are tainted because of the presence of a few sysPages (24 pBlocks contain only 1 and more than 90% of pBlocks contain less than 10% sysPages). It is evident from Table 1 that current anti-fragmentation framework of Linux is inefficient with respect to restricting kernel pages within few pBlocks.

3.2 Issues with Memory Compaction

Memory compaction, in its current form, does not interact with the buddy allocator[3] 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 pBlocks containing sysPages, free regions will not be accumulated at huge page granularity.

[3]Note that the buddy allocator can not help much when most pBlocks contain sysPages. It is the primary reason why compaction does not get target pages via the buddy allocator.

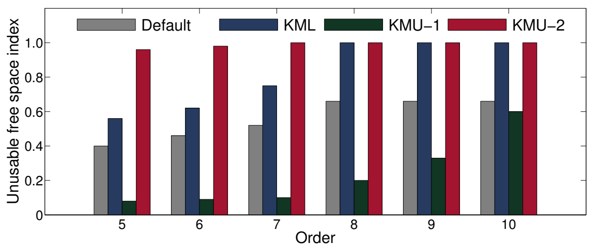

Figure 3: Unusable free space index (lower is better) with different kernel memory arrangements.

- If target pages are selected from pBlocks 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 pBlocks.

To investigate and validate the properties discussed above, we use two variants of an array-based bank buddy allocator [20], KML (kernel memory lower) and KMU (kernel memory upper) for placing sysPages 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, sysPages are aligned towards the lower end of memory while pBlocks 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 pBlocks containing sysPages in KML.
- In KMU, since source pages are selected from user pBlocks, it should be relatively easy to free contiguous regions for immediate use. However, target pages come from kernel pBlocks which should result in exacerbated fragmentation even for a few kernel memory allocations after compaction.

We can measure the success of memory compaction using the *unusable-free-space index* [11], for different memory block sizes as follows -

$$F_u(j) = \frac{TotalFree - \sum_{i=j}^{max} 2^i k_i}{TotalFree} \quad (1)$$

where *TotalFree* is the number of free pages, *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 3 shows the *unusable free space index* across all configurations. Note that KMU-1 and KMU-2 represents the KMU configuration immediately and some time after compaction. The experiments confirm our observations that the physical location of sysPages plays an important role in determining the success of memory compaction.

Another observation comes 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 the tainted pBlock 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 pBlocks is necessary but not a sufficient condition for controlling fragmentation.

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 important within their disciplines, a cooperative fragmentation management is required for a robust huge-page friendly memory manager. Unfortunately, it is not possible for us to cover all related aspects in this context. Hence, we focus on improving anti-fragmentation alone in the rest of the paper.

4. Optimizing Anti-Fragmentation

Facilitating huge page allocations on demand is difficult as long as kernel pages are present in majority of pBlocks. Hence, improving the effectiveness of page clustering with an optimized anti-fragmentation solution is the first and most critical requirement to control fragmentation at huge page granularity. We discuss few optimizations to improve page clustering in Linux kernel (version 4.1.13) in the following subsections.

4.1 Active Anti-Fragmentation

Active anti-fragmentation (AAF), inspired by Joonsoo Kim [13], has been discussed in the Linux community as a potential optimization for improving fragmentation framework of the Linux kernel. The idea behind AAF is to always reserve an entire pBlock during kernel memory fallbacks. If there are usrPages already allocated from it, they are moved elsewhere in memory while reserving the pBlock. This page

Algorithm 2 : Adaptive pBlock Selection

```
1:  $fallback\_kernel\_APBS(request\_order)$
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:      $page = get\_page\_AAF(USER, request\_order)$
9:  else /* Critical */
10:     $page = get\_page\_RPBS(USER, request\_order)$
11: end if
12: ....
13: return $page$
```

migration in AAF can potentially stall applications as pages being migrated cannot be accessed, also it can induce more TLB shootdowns due to page mapping being modified. It is prone to performance slowdown when workloads are aggressive (could be moving pages potentially at every request).

4.2 Optimal pBlock Selection

In section 2, we discussed a modified algorithm for selecting a pBlock which reserves maximum pages during fallback without migrating any pages. We call it *optimal pBlock selection (OPBS)* and its rate of pBlock mixing is shown in Figure 2. In OPBS, all eligible pBlocks are scanned during fallback to select the optimal pBlock. Even though the cost of scanning pBlocks 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 [5] have been implemented on top of the buddy allocator to serve memory requests of kernel objects. The buddy allocator handles kernel memory allocations only when slab pages get exhausted.
- Memory domains tend to balance their size depending on the workloads. Hence, the majority of kernel allocations come from kernel domain.

We observed less than 1% memory requests being served through fallbacks for more than 30 minutes of profiling for a kernel memory intensive workload. Hence careful optimizations here are unlikely to cause dramatic performance degradation in the overall performance of page allocator.

4.3 Adaptive pBlock Selection

In adaptive pBlock selection (APBS), we classify the state of memory with respect to fragmentation in four levels (i.e., low, medium, high, critical) for implementing kernel fallback routine in Adaptive pBlock Selection (APBS). Note that the buddy allocator starts lookup from the highest-order free lists during fallbacks (line 2 in Algorithm 1). Hence the level can be determined at runtime, depending on the order of the page being selected. We classify order 9 to 10 as low, 7 to 8 as medium, 4 to 6 as high and rest as critical fragmentation levels. Different callbacks used for each level are discussed below:

- Low: Memory is considered to be safe in this state as full pBlocks can be reserved during fallbacks hence default pBlock selection is sufficient.
- Medium: The best out of four random pBlocks is selected and used for allocation at this level.
- High: Along with reserving free pages, we also invoke AAF to move all usrPages out of the selected pBlock and add it to kernel domain.
- Critical: Fragmentation is severe and free memory is divided is small regions. Invoking AAF at this point incurs non-negligible overhead in the system. Hence, we use a controlled version of OPBS at this level to select the best of 64 pBlocks. We call it as Regulated pBlock Selection (RPBS).

Algorithm 2 shows the sample code snippet for APBS. At the end of two successive kernel builds, we find the number of tainted pBlocks in APBS configuration to be approximately 52% of the default kernel. OPBS further reduces the tainted pBlock count by 20%. The effectiveness of OPBS and APBS clearly shows that current anti-fragmentation suffers due to poor page selection during fallback in Linux. However, OPBS benchmarking reports some overhead in the runtime performance during kernel builds (2%-3% in system time). Our initial evaluation indicates that APBS is a reasonable fit for most systems (tested upto 16GB physical memory) in terms of restricting kernel memory within fewer pBlocks without noticeable runtime overhead (less than 1%). Note that the implementation of APBS is scalable for large memory systems as well since RPBS never scans memory beyond 64 pBlocks.

5. Evaluation

5.1 Test Setup

Our primary setup is a laptop equipped with an Intel® Broadwell processor with 2*2 cores operating at 2.3GHz and LLC (Last Level Cache) size of 3MB. The size of physical memory varies between 2GB and 4GB depending on the

Kernel	# Tainted pBlocks	Unusable Free Space Index
Default	1683	0.61
AAF	134	0.24
OPBS	508	0.21
APBS	874	0.31

Table 2: Unusable free space index for huge pages (order 9) and the number of tainted pBlocks with different kernels.

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

Table 3: TLB miss ratio for canneal and stream with (W HP) and without huge pages (W/O HP).

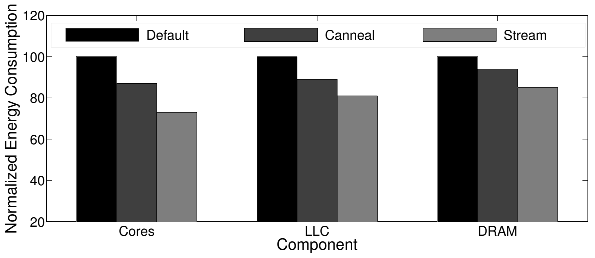

Figure 4: Normalized energy consumption of stream and canneal with additional huge pages.

benchmarks. *cpuid* reports L1 d-TLB and i-TLB supporting 64 entries for 4KB and 4 entries for 2MB pages while L2 contains 1536 entries for both 4KB and 2MB pages. The system provides robust support for measuring system-wide energy consumption across CPU cores, package (cores + LLC) and DRAM through hardware counters, which can be accessed via RAPL interface [19] available in *perf*.

5.2 Unusable Free Space Index

Table 2 depicts the number of tainted pBlocks as well as the *unusable free space index* for huge pages (i.e., order 9) on our system with 4GB physical memory. Note that though AAF performed better in terms of restricting kernel memory within fewer pBlocks, 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 pBlock selection was aligned towards KML configuration discussed in Section 3 during this test. It also confirms our earlier observation that minimizing tainted pBlock count is not sufficient for controlling fragmentation. APBS performs reasonably well in term of both restricting fallbacks to few pBlocks as well as reducing the *unusable free space index*.

5.3 Benchmarking Huge Page Allocations

In an another experiment, we use *stress-highalloc* from *mmtests* [9] 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%, APBS was able to allocate 7% of system memory as huge pages. When system was at rest in the last attempt, APBS was able to satisfy 37% of huge page allocations as compared to 13% of default kernel. Additional success rate of APBS was found to be varying between 25% and 30% with 4GB physical memory as well.

Benchmark	**Runtime (milliseconds)**			**Throughput (ops/m)**		
	W/O HP	**W HP**	**Gain**	**W/O HP**	**W HP**	**Gain**
stream	1549	1237	20%	x	x	x
canneal	90360	78940	13%	x	x	x
compress	x	x	x	127	134	5.5%
crypto.rsa	x	x	x	204	214	4.9%
derby	x	x	x	266	282	6.0%
sunflow	x	x	x	49	51	4.0%

Table 4: Throughput and runtime performance gain (wherever applicable) for stream, canneal and a few SPECjvm2008 benchmarks with additional huge pages.

5.4 Performance/Energy Measurement

25% additional success rate yields more than 450 huge pages on a 4GB system. We benchmark the benefits of these pages, using *libhugetlbfs* library support available for Linux systems, with some real applications. 450 huge pages were reserved before running the test applications which include *stream* [16], *canneal* (from PARSEC benchmark suite [4]) and a few *SPECjvm2008* [1] benchmarks.

We measure the impact of additional huge pages on the system TLB with canneal and stream benchmarks. TLB miss ratio for both applications and reduction in TLB miss ratio with huge pages is shown in Table 3. Interestingly, huge pages reduce TLB miss rate of canneal by up to 97% and up to 72% for stream. 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 4. Stream reports up to 20% reduction in runtime with default input array size, while canneal benefits up to 13% with native input set. Throughput gains for SPECjvm applications are also substantial (up to 6%).

Figure 4 shows normalized energy savings across cores, LLC and DRAM for our test applications. Our analysis shows energy gains to be more significant as compared to performance, which further adds to the motivation for better fragmentation management. For example, a performance gain of 20% in the case of stream translates into 27% energy reduction. Further analysis shows that both run-time overhead and energy cost of accumulating huge pages is less than 2% for these benchmarks. For long-running applications, the cost of page migration turns out to be insignificant compared to the overall savings.

6. Related Work

Navarro [18] proposed a reservation based scheme to support huge pages. Their approach is to restore fragmentation rather than preventing it beforehand, thus incurring management overheads. The approach is also unable to handle compile based workloads and is easily defeated in short time span.

Applications with fewer kernel memory will work well with their approach but fails miserably with workloads having larger kernel footprint.

Gorman [10] 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. J. Kim [14] presented a proactive anti-fragmentation approach that groups pages with the same lifetime, and stores them in fixed-size contiguous regions. The regions are reused by subsequent applications when a process gets killed. However, their approach is limited to mobile systems where killing an application is considered reasonable to recover from memory pressure. In general purpose computers their approach is not practical as killing a process is unacceptable to reclaim memory.

7. Conclusion

In this paper, we discussed the interaction of kernel page placement with memory compaction and explored some optimizations with respect to controlling fragmentation at huge page granularity. We also evaluated the benefits of huge pages with real applications. However, memory fragmentation is far from a solved problem yet and more sophisticated solutions need to be developed for a huge page friendly memory manager. A few potential solutions could be optimizing the existing anti-fragmentation and memory compaction framework. Reevaluating kernel design itself against its inability to control fragmentation [6] is another viable option (for example, redesigning kernel to facilitate movable kernel pages [8] is an interesting topic on its own). Recent research (i.e., Prudence [21]) reports some interesting optimizations in the kernel memory allocators to control its memory footprint which could prove to be vital for fighting with memory fragmentation.

Acknowledgments

We would like to take this opportunity to thank our shepherd Prof. Gernot Heiser and the anonymous reviewers for giving us valuable feedback on the paper. We would also like to thank Aravinda Prasad, Sandeep Kumar and Priyanka Singla for their help in improving the quality of the paper.

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] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. 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. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485943. URL `http://doi.acm.org/10.1145/2485922.2485943`.

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

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

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

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

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

[9] M. Gorman. Mmtests: Benchmarking framework primarily aimed at linux kernel testing. URL `https://github.com/gormanm/mmtests`. https://github.com/gormanm/mmtests.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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. doi: 10.1145/2754169.2754179. URL `http://doi.acm.org/10.1145/2754169.2754179`.

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

[16] 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.

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

[18] 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`.

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

[20] 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. doi: 10.1109/HiPC.2015.42. URL `http://dx.doi.org/10.1109/HiPC.2015.42`.

[21] 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. doi: 10.1145/2872362.2872405. URL `http://doi.acm.org/10.1145/2872362.2872405`.

[22] 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/.