Prevelence of Page Splintering

Akash Panda[1]

***Problem Statement*— Translation Lookaside buffers(TLB) misses are long latency operations. In hypervisor-based virtual environments, the penalty is even worse as TLB miss would induce a nested page table walk. Huge pages (2MB or 1GB) is supported by modern architectures to curtail the overhead due to TLB misses. An increase in page size would mean less number of page table entries. KSM is a memory deduplication technique, which improves efficiency of memory usage. With huge pages in picture, the probability of two pages being exactly equal would be very less. So, there is a tradeoff between access performance and deduplication rate. Host operating systems may splinter pages in order to achieve good memory efficiency. My work is to calculate the amount of Page Splintering present while modern workloads are being run.**

I. Introduction

A. Motivation

Page table is the data structure that holds the Virtual Address to Physical address mappings. At the beginning, if a processor needs to map a virtual address to a physical address, it would traverse the page directory completely to get the page table entry of interest. Hence, Using a typical 4 level page table walk in order to access a memory location, we make 4 memory references just to get the physical address of the memory location to be accessed. Here comes TLB to the rescue. TLB takes into account that most processes exhibit a locality of reference[2]. TLB is a small associative memory that caches the Virtual address to Physical address mappings. A TLB miss would incur a long latency operation, which would require a page table walk to get the address mapping. If we look at a hypervisor based Virtual machine environment, the TLB miss penalty is enormously high as a two-dimensional page table walk would incur 24 memory references as compared to 4 in non virtualized environments.

Buell *et al* looked at areas that made applications run more slowly in virtualized environments. They found out that increase in TLB miss handling penalty in the hardwire assisted Memory Management Unit is the largest contributor to the performance gab between native and virtualized servers.[1].

Large pages improves memory access performance. Large pages implies fewer page table entries and a larger TLB reach. Fan Guo *et all* has shown that enabling large pages in both guest and host can improve memory access performance by up to 68% [3]. Redundant data is present across Virtual machines, where many of the pages across VMs share the same content. Base sized pages (4KB pages) have a high probability to share the same content with another base sized page. The probability of two huge pages sharing same content is quite less. Fan Guo *et all* have shown that large pages also reduce the deduplication opportunities [3]. Virtualization systems often splinters the guest operating system's large pages into base size pages in the host, sacrificing the performance benefits.

With emergence of cloud computing, virtualization technologies are being hugely employed in industry to cater diverse workloads. As already mentioned, the use of large pages in both guest(virtualized) system and host system can significantly increase the performance of the application running in the virtualized servers, whereas splintering of large pages in host system may sacrifice performance benefits.

If we can know what amount of pages are being splintered and what is the main cause behind splintering, we can then start looking at solutions to take benefit of both reduced address translation overhead and page deduplication engine efficiency.

B. Achievement

I evaluated amount page splintering happenning while running different workloads in virtualized environment.

II. BACKGROUND

A. Virtual Memory

The x86 architecture allows for more memory to be addressed than is available physically in the hardware. It is achieved by having each process access its own addressable memory. The process thinks the whole memory is available for its use. This is known as the virtual memory of the process.

B. Page table

Page table is the data structure that holds the Virtual Address to Physical address mappings, where the data is actually stored. Page table consists of Page table entries(PTEs), which stores a frame number and optional status (like protection) bits. Address space is always sparsely populated, as most of the processes do not use the full available address space in 32 bit systems and not even a part of it in 64 bit systems. Hence, the page table is implemented as a sparse tree representing the address space.

C. Two-dimentional page tables

Guest Virtual Address(GVA) are converted to Guest Physical Address(GPA) by unmodified guest page table. GPA is converted to System Physical Address(SPA) by nested/extended page table. Nested Paging along with native paging constitutes the two-dimensional page walk.

[1]M.Tech(Research), Department of CSA IISc Bangalore, India akashpanda@iisc.ac.in

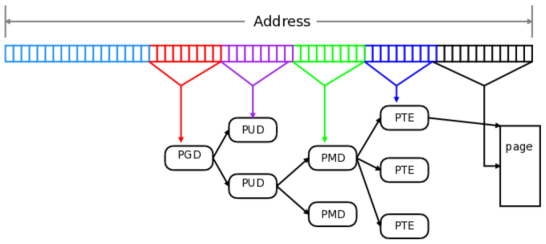

Fig. 1. Virtual Address to Physical Address [5]

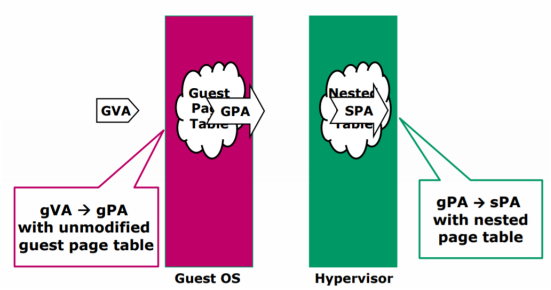

Fig. 2. Nested Paging [4]

D. Large Pages

As we already know, memory is managed in blocks known as pages. Size of a regular page is 4KB. There are two ways to manage large amount of memory - (a) Increase the number of TLB entries, (b) Increase the Page size.

First method is inefficient, as current hardware only supports limited number of TLB entries. Also the algorithms for the hardware and memory management may work well with thousands of entries but many not perform well with millions of entries.

Blocks of memory of 2MB or 1GB sizes are called huge pages. The page tables used by the 2MB pages are suitable for managing multiple gigabytes of memory, whereas the page tables of 1GB pages are best for scaling to terabytes of memory. Huge pages can be difficult to manage manually, and often require significant changes to code in order to be used effectively. Linux kernel implements Transparent Huge Pages(THP). THP hides much of the complexity in using huge pages from system administrators and developers. THP is an abstraction layer that automates most aspects of creating, managing, and using huge pages. THP support is an optimization. Large pages are managed automatically and transparently for the application.

E. Kernel Samepage Merging(KSM)

KSM is a page deduplication feature. It is a scanning based mechanism to detect and share pages having same content. KSM is implemented as a linux kernel thread that runs and periodically scans the memory regions, advised as mergeable(by calling madvise(MADV_MERGEABLE)) looking for identical pages. When identical pages are found, it merges them and marks it copy on write(COW). KSM is able to merge only Anonymous memory and not memory mapped pages. As memeory of guest is black box to the system, so all the pages are mapped as anonymous. Hence, KSM can merge all of the pages related to a Virtual Machine. [6]

KSM uses two red-black tree data structures for lookup: stable and unstable trees. All merged pages are being held by stable tree's pointers, while all potential sharing candidates are being held by unstable tree's pointers. A potential sharing candidate is a page which did not change from the last time it has been compared. The trees are sorted by the content of the pages. The whole KSM process can be seen in Figure 3.

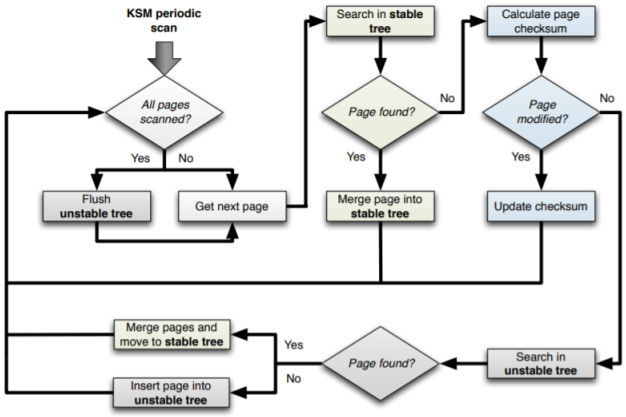

Fig. 3. KSM process [6]

KSM allows memory pages with identical content to be transparently shared between Linux processes, and therefore between Linux virtual machines.

F. Large Pages and Deduplication Opportunities

Redundant Data is common across Virtual Machines. Many of the 4KB pages have same contents. But large pages reduce deduplication opportunities. Very few large pages are exactly the same. Deduplication may not be useful in case of large pages. Aggressive Deduplication Approaches are currently employed by OSes. They split the large pages into base pages aggressively and performs deduplication among base pages [3]. Although this saves significant amount of memory, but the page tables grow bigger, which will in turn reduce the hit ratio of TLB and increase the address translation time.

III. Achievements

A. Implementation

I used Linux Kernel (v4.19.0) as the base version on which I performed my experiments. I wrote a kernel module to get information about pages of a process. When the kernel module is loaded, it would dump the information about the pages of the particular process (whose pid is to be given as an argument to the module).

Code snippet for the above process is shown below:

```
struct mm_struct *mm = pid_to_mmstruct(pid);
if(mm == NULL) {
    //Report Error to the module
```

```
    return -1;
}
struct vm_area_struct *vmas = mm->mmap;
while (vmas != NULL) {
    unsigned long vm_start,
    vm_end;
    vm_start= vmas->vm_start;
    vm_end = vmas->vm_end;
    while(vm_start <= vm_end) {
        int page_type;
        unsigned long pfn_value =
            get_pfn_value (
            mm , vm_start , &page_type
            );
        //Dump the Page information
        if(page_type == PT_1G) {
            vm_start +=
                ((unsigned long)1 << 30);
        } else if(page_type == PT_2M) {
            vm_start +=
                ((unsigned long)1 << 21);
        } else if(page_type == PT_4K){
            vm_start +=
                ((unsigned long)1 << 12);
        }
    }
    vmas = vmas->vm_next;
}
```

I wrote a kernel module to get guest Page Frame Numbers to Host Virtual Addresses. The input to this module will be file name of the file, where the Guest Page Frame numbers are dumped into and the pid of the running qemu process which holds the Virtual machine.

Pseudo code snippet of the above process is shown below.

```
// Get file statistics in stat
// Then get size of file
unsigned long long size_of_file
    = stat->size ;
// Allocate data as per size of file
char *data = kmalloc(
    sizeof (char) * size_of_file ,
    GFP_ATOMIC
    );
unsigned long long no_of_vas = 0;
if(!data) {
    //Inform the module about
    //failed allocation of space
    //to read data.
} else {
    file = file_open(
        filename, O_RDONLY , 0
        );
    file_read(
        file, 0 , data, size_of_file
        );
    char *virtual_address_string;
    // loop all lines in the file as line
        //virtual_address_str = line
        //Dump mapping using
                // gfn_to_hva(
                //  get_kvm_ptr(pid) ,
                //  virtual_address);
    }
}
```

I wrote a script to invoke the kernel modules and generate the statistics. The flow of experiment is shown in Figure 4.

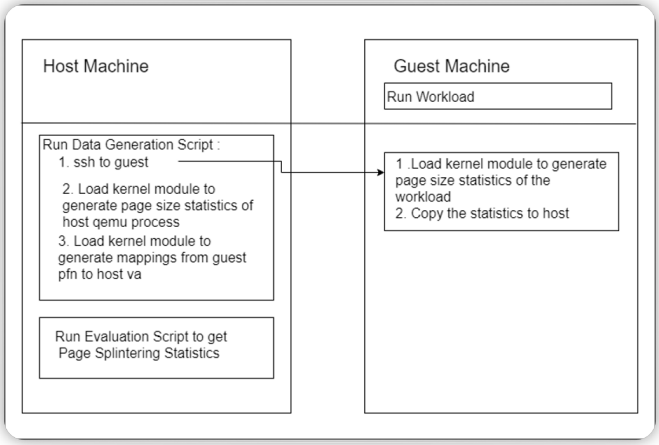

Fig. 4. Flow of code

I ran the workload in the guest Operating System. Then would run the Data and Statistics collector script at the host, which would ssh to the guest machine to generate the information of pages of the guest, and would copy the statistics to host. Then it would generate statistics at host system for the pages of the qemu process. It would then generate the Guest Physical to host virtual mapping. At last it would parse the generated data to get the splintering information.

B. Experimental Setup

CPU: Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz

Memory: 32 GB

Hypervisor: KVM

Virtual Machine's memory: 10 GB

The experiments were conducted enabling THP at both guest and host. The enabled setting of THP was kept "always".

C. Workloads used

Workload	Description
XSBench	XSBench is a mini-app representing a key computational kernel of the Monte Carlo neutronics application OpenMC.
Gups	The RandomAccess benchmark as defined by the High Performance Computing Challenge (HPCC) tests the speed at which a machine can update the elements of a table spread across global system memory, as measured in billions (giga) of updates per second (GUPS).
Redis	Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker.
Aerospike	Aerospike is a distributed, scalable NoSQL database.

IV. Results

Figure 5 and Figure 7 shows the prevalence of page splintering in those workloads. Figure 5 shows the percentage of huge pages allocated in those workloads and Figure 7 shows the percentage of huge pages allocated in guest being splintered into small base size pages in host. Percentage of huge pages is measured as the total number of huge pages out of the total number of pages. Percentage of Memory backed by huge pages is the amount of memory being backed by huge pages out of the total memory of the process. Percentage of page splintering is measured as percentage of large pages in the guest being broken down into base sized pages in the host.

Workload	Percentage of Huge pages
XSBench	57.25
Gups	45.38
Redis	0.70
Aerospike	33.83

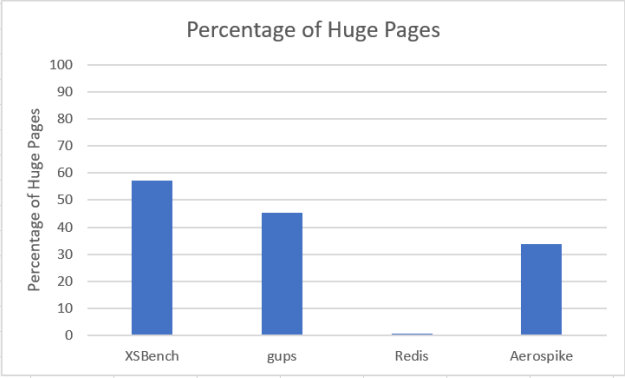

Fig. 5. Percentage of Huge Pages

Workload	Percentage of memory backed by Huge pages
XSBench	99.85
Gups	99.76
Redis	77.47
Aerospike	99.26

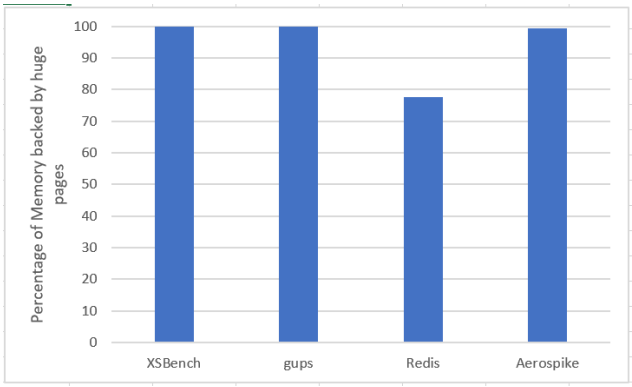

Fig. 6. Percentage of Memory backed by Huge Pages

Workload	Percentage of Page splintering
XSBench	66.59
Gups	58.92
Redis	90.57
Aerospike	95.36

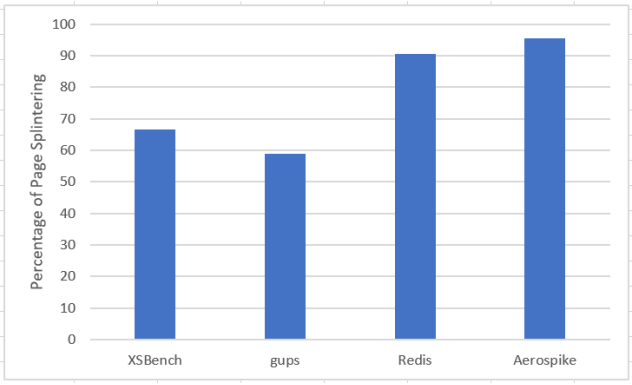

Fig. 7. Percentage of Page Splintering

V. Conclusion and Future Work

From the results, we are able to see that there is a quite good amount of splintering while running these workloads. XSBench had 66.58% of huge pages being splintered in the host, gups had 58.92%, redis 90.57%, and for aerospike 95.35% large pages are being splintered into base pages.

Future work can be carried out on exploring the reason for splintering in each of the workloads, and find out a solution to get balance between the page deduplication benefit and large page benefits.

VI. Source code

Kernel modules used in the project can be found at: https://github.com/akashiisc/kernel-modules.git

Modified kernel used in the project can be found at : https://gitlab.com/akashpanda/kernel.git. You have to check-out the splintering-4.19 branch.

Sciripts used for statistics collection can be found at: https://github.com/akashiisc/statistics-collector.git

References

[1] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, H. Reza Taheri. Methodology for Performance Analysis of VMware vSphere under Tier-1 Applications. VMWare Technical Journal, Summer 2013, pp19-28.

[2] https://www.kernel.org/doc/gorman/html/understand/understand006.html

[3] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, John C. S. Lui. SmartMD: A High Performance Deduplication Engine with Mixed Pages. In the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)

[4] http://www.linux-kvm.org/images/c/c8/KvmForum2008$kdf2008_21p̀df

[5] https://lwn.net/Articles/716603/

[6] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, Frank Bellosa. KSM++: Using I/O based hints to make memory deduplication scanners more efficient. In the proceedings of RESoLVE'12.

[7] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge

[8] https://www.perftuning.com/blog/linux-hugepages-improves-x86-memory-performance-large-databases/

[9] https://lwn.net/Articles/330589/

[10] https://wiki.debian.org/Hugepages

[11] https://www.aerospike.com/docs/

[12] https://redis.io/

[13] https://github.com/alexandermerritt/gups

[14] https://github.com/ANL-CESAR/XSBench