Hzmem: New Huge Page Allocator with Main Memory Compression

Guoxi Li, Wenzhi Chen, Kui Su, Zhongyong Lu, and Zonghui Wang

College of Computer Science
Zhejiang University, Hangzhou, 310027, China
{guoxili,chenwz,sukuias12,lzy6032,zhwang}@zju.edu.cn

Abstract. Today, applications that require large memory footprint prevail in cloud computing fields from both industry and academia. They impose great stress on the memory management of operating system, spend quite a substantial proportion of time dealing with TLB misses and excessively reduce consolidation ratio in term of server consolidation and virtualization. There are two methods to address these problems: main memory compression and large page support. However, to the best of our knowledge, there is no existing practical research on combination of these two methods since the combination in commodity operating system requires lots of low-level design modifications.
We propose a new memory management framework that is decoupled and flexible for easy developments and is able to run simultaneously with the original memory management. On top of the new framework, we implement Hzmem, a completely new large page memory management redesign with features of main memory compression to address the aforementioned problems once for all but requires only minor modifications to the other subsystems of the underlying operating system. Our method achieves competitive performance compared with native large page support, increases effective memory size and impacts little on other subsystems of operating system.

Keywords: Large page, Main memory compression, Linux

1 Introduction

Nowadays, with more and more applying of cloud computing both in business area and research community, workloads are very likely to consume more memory than a single physical machine can offer. These memory-hungry workloads often require large memory footprint but show poor temporal locality[13, 6, 2]. They often involve relational databases, key-value stores and huge gateway machines handling huge routing data.

All above workloads require memory overcommitment to fulfill tasks in such situations. One kind of memory overcommitment like swapping can swap out memory presumably not to be used in the recent future onto disks. However, disk accesses are far slower than memory accesses, bringing lots of overheads. Thus

developers and researchers resort to another practical memory overcommitment technique—main memory compression. Main memory compression compresses memory and stores compressed data in reserved memory regions, enormously increasing effective memory size and avoiding disk access latencies.

In the meantime, traditional physical memory management with granularity of constant size lacks its meaningfulness and efficiency. Current commodity operating systems, like Linux and FreeBSD, have introduced their own large page (memory page size larger than base page) supports. They are not self-contained mechanisms, but are heavily dependent on the traditional memory management on base pages.

We consider such problems in the context of modern commodity operating system like Linux, but the ideas are not specific to the Linux. Currently, the Linux `hugepage` (Linux terms for large page) mechanism demands 4 KB base pages from the famous `Buddy System` and merges them to form a larger size (2 MB on x86-64 platforms), which is complicated and time-consuming. Since hugepage mechanism stems from traditional base page managements, frequent and heavy `hugepage` allocations would fail in most cases due to the lack of enough physically continuous memory especially after a long time of running. Moreover, `hugepage` in Linux does not support main memory compression and it is difficult to add main memory compression feature since that needs heavy modifications to low level design of memory management which brings considerable effort.

Therefore, it has motivated us to redesign a completely new memory management framework that is decoupled and flexible for convenient development. Based on the new memory management framework, we implement Hzmem that includes a self-contained hugepage allocator which is decoupled from the normal page allocator in the environment of frequent and heavy hugepage allocations and easily equips the new hugepage allocator with main memory compression feature. It achieves competitive performance over native large page supports, increases effective memory size and impacts little on other subsystems of operating system.

Hzmem's benefits are 1) the new hugepage allocator has competitive performance compared with native hugepage implementations; 2) performance isolation, which means new allocator brings no or little performance punishment to other parts of system utilizing normal physical memory management; 3) increasing effective memory capacity, which increases consolidation ratio and improves performance when applications require memory beyond the capacity.

2 Related Work

We briefly discuss some works related to our work.

Memory mangement architecture: Recently, with trends of workloads using memory quite differently from the time when memory management was designed, lots of researches[2, 3, 9, 7] are focused on modern memory management.

Basu et al.[2] propose mapping part of a process's virtual memory address with *Direct Segment*, removing TLB miss penalty. *Direct Segment* maps a contiguous virtual memory directly to a contiguous physical memory using simple hardware requirements. However, *Direct Segment* needs both software and hardware supports, making it not suitable for commodity hardware.

Clements et al.[3] propose a new virtual memory system design called RadixVM that removes serial operations on virtual memory and enables fully concurrent operations by ensuring non-overlapping memory regions. RadixVM need so many modifications to a commodity operating system that it is only implemented on a Unix-like teaching operating system.

Huang et al.[9] conduct a comprehensive and quantitative survey on the development of the Linux memory management over five years (2009-2015). The study shows the changes and bugs are highly centralized around the key functionalities, like memory allocator and page fault handler.

These studies give many insights and lessons that modifications to commodity operating system's key functionalities are very challenging and our work manages to avoid the challenges by using a decoupled and flexible framework.

Huge page support: Navarro et al.[12] implement OS support for large pages in FreeBSD. They focus on reservation-based allocation and fragmentation control. Hzmem is built on a decoupled and flexible memory framework which is detached from the normal base page allocator. Therefore, fragmentations of base pages impact little on Hzmem.

Kwon et al.[10] propose Ingens, a framework for transparent huge page support through tracking utilization and access frequency of memory pages. In contrast to Ingens, Hzmem's focused on adding main memory compression feature in order to increase effective memory size and improve consolidation ratio in terms of server consolidation and virtualization.

Main memory compression: Ekman et al.[5] propose a main memory compression framework that eliminates performance losses by exploiting simple and yet effective compression scheme, a highly-efficient compressed data locating and a hierarchical memory layout. Pekhimenko et al[14] propose *Linearly Compressed Pages* (LCP) that avoids the performance degradation problem without requiring costly or energy-inefficient hardware. Like these two work, Hzmem also makes optimizations on zeroed pages yet with no need of any hardware modifications.

Tuduce et al.[15] propose a main memory compression solution that adapts the allocation of real memory between uncompressed and compressed pages. It allows to shrink or grow the size of the compressed area without user involvement and it is implemented in Linux over commodity hardware. In contrast to this work, Hzmem is focused on the compression of large pages and stores the compressed data in base pages. This will lead to negligible wasting space at a percentage of no more than 1/512.

3 Motivation

Most modern commodity operating systems support large pages. For example, Linux allows applications to use specific API (it is called `hugetlbfs` in Linux) to allocate memory based on large pages (`hugepage` in Linux). These large pages are allocated from memory pools that are preserved in advance by administrators. Moreover, these memory pools are in turn allocated from Linux `Buddy System`.

What leads to this lengthy detour on implementation of large page memory management? It is that the memory management design in Linux inherently bases on the fact that the page size is constant (4 KB in Linux). Linux uses the macro `PAGE_SIZE` to represent this base page size, which is used throughout almost all the Linux subsystems, like virtual memory management, physical memory management, I/O subsystem, page reclaiming, etc. For example, I/O subsystem assumes a fixed page size as 4 KB and this goes well with the 4 KB block size that is multiples of a sector size. As for large page support, it is around in Linux since 2003[4] when it has been long after Linux was designed and developed and some assumptions of design cannot be modified easily.

Consequently, Linux resigns itself to adding large page feature upon the base page memory management though overheads and maintainability problems will be caused. For example, a large page is treated as 512 contiguous base pages in an aligned 2 MB region and still uses the same page descriptors (`struct page` in Linux which holds meta informations for one page). However the page descriptors for large pages are page descriptors for base pages linked together as `compound pages` with heads in linked lists holding useful information, which is a great waste of memory space. As one of the most used data structures in kernel, the page descriptor in Linux has to meet requirements from many subsystems and thus this largely increases its size. Since there are so many page descriptors that a single byte increase will lay much stress on kernel memory use.

Linux has already had supports for main memory compression like `zram, zswap and zcache`, but currently these supports are highly dependent on page reclaiming subsystem. Page reclaiming subsystem is an important part in Linux which also assumes a fixed page size. It mainly contains two reclaiming procedures that are 1) writing back pages that are backed up by files and 2) swapping out pages that are not backed up by any persistent storage devices. Both these two processes involve the aforementioned I/O subsystems that also assume a fixed page size. For example, `zram` treats itself as a block device that is used as destination for swapping.

Equipping large page support with data compression feature based on these compression techniques will cost non-trivial efforts to modify Linux and even question some assumptions Linux has long held. Moreover, large page supported through `hugetlbfs` has no backing up storage devices and there is no need for swapping or writing back to persistent storage devices. Thus it is not practical to enhance Linux with data compression feature using current compression supports and may bring instability to Linux base code. Therefore, in order to meet our requirements, we need to redesign the physical memory management.

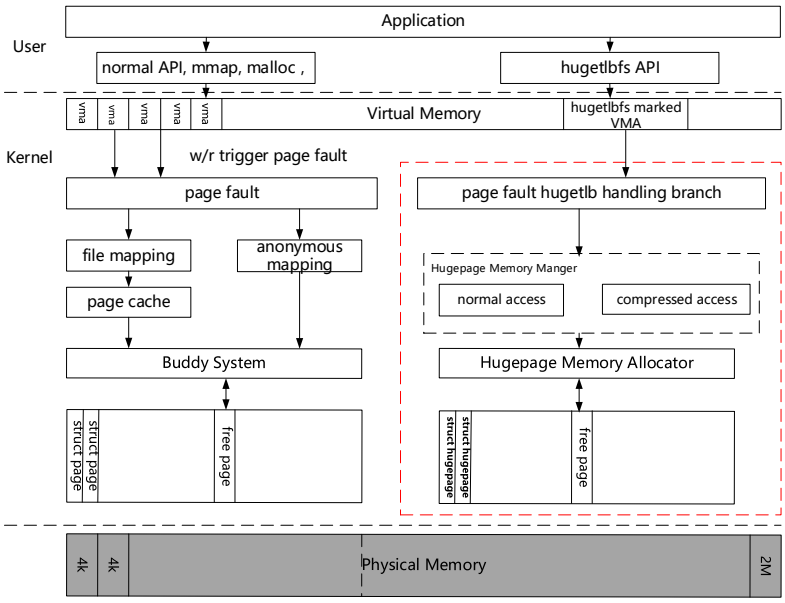


Fig. 1. Architecture of decoupled and flexible memory management framework

4 Architecture and Implementation

We implement Hzmem based on the decoupled and flexible memory management framework with a completely new hugepage allocator, a feature of main memory compression and `hugetlbfs` API compatibility. The decoupled and flexible memory management framework is shown in Figure 1. The new framework contains a placeholder for a physical memory manager that can be implemented according to academic or industrial requirements and run with the original one at the same time, thus making it decoupled and flexible. Figure 1 only takes Hzmem for example in that placeholder. Applications in user space using normal API or `hugetlbfs` API to create virtual memories which will be translated into different physical memories managed by different manager respectively. Hzmem contains four components: physical hugepage memory management, page fault handler for hugepages, page reclaiming for compressing hugepages and hugepage compression data management. Figure 2 shows workflow among four components. A daemon in page reclaiming module checks allocated huge pages in every NUMA node, selects the cold ones and invokes compression interfaces in compression data management. It also changes the page table accordingly, which makes it possible to retrieve the compressed huge pages. When an application accesses the compressed data, a page fault is triggered. The page handler identifies the compressed huge page, invokes decompression interfaces and restores the page table entries. Finally the application can resume and access the data.

In a word, page reclaiming and page fault handling work in reverse ways. Their cooperation makes the framework function properly.

We implement 4148 lines of C code (LoC). It runs over Linux with kernel 3.10, functioning simultaneously well with the original Linux memory management subsystem.

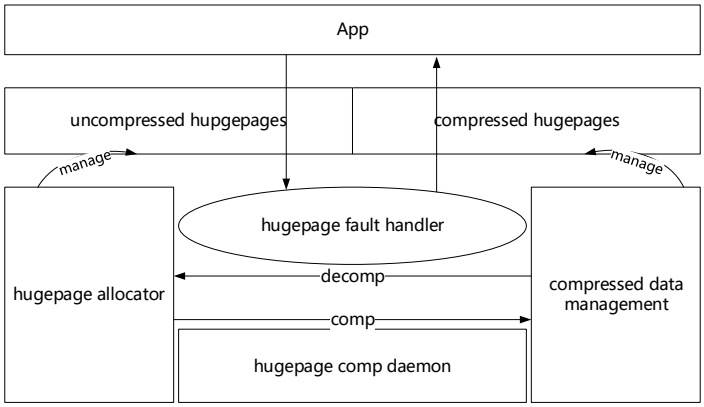


Fig. 2. Workflow of Hzmem

4.1 Hugepage Physical Memory Allocator

We take a clean-slate approach to implement the hugepage physical memory allocator from the ground up.

New Page Descriptors The new memory management is specialized for 2 MB huge pages in x86-64 architecture (currently we do not support 1 GB large pages) and only used by user level applications through `hugetlbfs` API. In the new framework, we drop the original method or workaround that combines 512 contiguous page descriptors into a compound page descriptor for a huge page. Instead, we use customized page descriptors specific only to huge pages that are simple and without any "compound" and space-wasting problems. This will significantly save memory space used for storing page descriptors since both the number and the size are decreasing. For example, a modern commodity server with memory capacity of over 256 GB using the new page descriptors can save several GBs which is 1%-2% percent of the memory and the situation worsens when larger memory available.

Without any need of concerns on other subsystems, our new page descriptor `struct hugepage` only deals with the specific use cases for huge pages used through `hugetlbfs` by use level applications. This makes the new descriptor

deprive itself of the fields of structure related to `slab`, `compound page`, etc., which is very suitable for use cases where memory is under severe pressure that it is resigned to main memory compression.

Free and Allocated Page Management With large memory capacity, each node in NUMA system can reach tens of GBs. To maintain such large memory, one node cannot be managed like that of Linux: except the first node, a node contains only one zone called `ZONE_NORMAL`. In our memory management framework, memory of one node is divided into several sections according to its size. Currently, we set size of one section to be 4 GB heuristically. It enables us manage memory in a smaller granularity and achieve better scalability and parallelism.

In Linux `Buddy System`, the highest order of contiguous memory is 10. It is contiguous memory of up to 4 MB that Linux is designed to manage. A huge page is 2 MB of order 9 that is almost the highest. Therefore, maintaining free huge pages in higher order will not bring much benefits and thus we keep free huge pages in a linked list in each section one by one. It deprives us of splitting and coalescing neighboring memory, thus accelerating allocation and deallocation speed.

During allocation, the allocator moves one free huge page from the free list into a new linked list called `lru list`. Every node has two `lru lists` that are used for holding allocated pages: one active list for hot pages and the other for cold pages, which makes it convenient for page reclaiming described in Section 4.3. During deallocation, the allocator can get information of node and section from the new page descriptor which helps put the huge page back to the proper free list.

Initialization Our hugepage physical memory allocator should be initialized at the same time of other unmodified memory management subsystems like `Buddy System`.

First, we reserve a range of contiguous physical memories that are to be detached from the management from `Buddy System`. After operating system finishes booting, we have 1) `Buddy System` manage memory of base pages mainly for kernel memory and user level applications' sections of base pages; 2) the new hugepage physical memory allocator manage the reserved memory for huge pages that are used by applications through `hugetlbfs` API.

We make minor modifications to Linux code base. We add hooks to the architecture-specific memory detect during booting and mark a range of memories as reserved, making them invisible to `Buddy System`.

Finally, as is ignored by operating system, we set up the direct mapping for the specific range of memories on our own, making it convenient to access the memory without triggering page faults in kernel mode.

4.2 Page Fault Handler

A page fault is called "soft" page fault when it merely allocates a new page and sets up a new page table entry, without reading contents from backing physical store devices like disks. Huge pages are not backed up by persistent storage. Therefore, what we focus on is the soft page faults triggered by huge pages.

A user level application allocates huge pages from our new memory management through compatible `hugetlbfs` API. After mounting `hugetlbfs` filesystem, creating a file at the `mount` point and calling `mmap` system call on the file, an application is able to take advantage of our new huge page memory management by accessing the mapped memory. It is the `mmap` system call that marks the range of virtual memory as "`MAP_HUGETLB`" which enables us identify what needs to be allocated through the new hugepage memory allocator in page fault handler.

We replace the old hugepage code path in page fault handler with our new code path based on the new hugepage memory allocator. Since the page fault related to huge pages is the soft page fault and has the feature of compression, there are two situations:

1. Normal page fault case where the physical page is accessed for the first time. Since the huge page fault is a soft page fault, the page fault handler just allocates a zeroed huge page from the new hugepage physical memory allocator.
2. Page fault case where the physical page is protection violation. It is either a shared page that can be retrieved from page cache or a compressed page that can be decompressed and reclaimed from the compression data mangement subsystem described in Section 4.4

4.3 Page Reclaiming

Page reclaiming monitors and identifies the allocated huge pages as cold or hot pages. Cold pages are isolated to be ready for reclaiming. In every node of operating system, there is one daemon called `hp_kscannerd` that does periodical checks on the usage of huge pages. We take lessons from Linux `Buddy System` that every node has a watermark indicating whether the memory usage of the node is under pressure. If the number of free huge pages is below the watermark, the daemon on the corresponding node wakes up and starts reclaiming pages identified as cold through compressing interface of compression data management described in Section 4.4 and changes the page table entries accordingly.

We use the second chance algorithm taken from original Linux page reclaiming mechanism to identify an allocated page as cold or hot. There are two states involved in page descriptor: active and referenced in the page descriptor; and one state in page table: page accessed bit in page table entry. Referenced state indicates whether the page is accessed and active state indicates whether the page is in active list. Whenever the physical page is accessed the bit in page table entry is set by hardware without any operating system interferences. Software is

responsible for clearing the bit periodically and setting referenced state accordingly in page descriptors. The waked daemon scans `lru lists` and checks the referenced state to determine whether the page should be reclaimed: 1) only two consecutive referenced state sets or clears can cause the page to be transferred between the active list and the inactive list; 2) the pages in inactive list are ready for reclaiming.

4.4 Hugepage Compression Data Management

Hugepage compression data management participates in controls of compression/decompression and compressed data management. It is the lowest level part as it provides compression/decompression interfaces for other parts to invoke. It is also important part as the speed of compression/decompression and efficiency of compressed data management impact greatly on performance of the whole system.

Compression algorithms. There are various kinds of lossless data compression algorithms. We choose LZO and LZ4 algorithms (the insight of choosing is out of scope of this paper). LZO algorithm appears since Linux kernel 3.10 and LZ4 algorithm since Kernel 3.15. We port both algorithms to our system. LZO is better than LZ4 in compression ratio and compressing speed, but LZ4 is better in decompressing[16] which we believe more important in memory management framework.

Optimization on zeroed huge pages. As mentioned before, when first accessed, the allocator just allocates zeroed huge pages, which account for substantial proportions of memory. When zeroed huge pages get compressed, they still occupy a good amount of memory. Thus, we optimize zeroed huge page compression through setting its size to 0 in compressed data region. This optimization, we believe, will achieve good improvements both in space and time when zeored huge pages are pervasive.

Compressed data management. Compressed data cannot be stored in memory backed up by huge pages. Since one compressed huge page is usually smaller than 2 MB, storing in huge page memory will waste a lot of space and make it difficult to manage the compressed data. The new allocator runs simultaneously with the `Buddy System`, which means there are two memory management mechanisms taking charge of memory of different granularities in one machine. Taking advantage of this, we split the compressed data and store them in multiples of 4 KB blocks in physical memory space backed up by base pages.

In Linux we use the mature `vmalloc` to allocate memory from base pages. There are two reasons: 1) simple, robust and virtual space for `vmalloc` is large enough in x86-64 architecture; 2) one compressed huge page will at most waste 1/512 space which is small and acceptable. Thus we create a red black tree and

use the `hugetlbfs` file along with index of huge pages as key for retrieving when decompressing is triggered in page fault handler.

5 Evaluation

We evaluate Hzmem using a variety of user applications and benchmarks, comparing against the performance of Linux's `hugetlbfs` support which is state-of-the-art. Experiments are performed on one machine with 16 Intel Xeon E7520 1.87 GHz CPUs and 64 GB memory. We use Linux 3.10 and Centos 7 for the host environment and use 4 KB for base pages and 2 MB for large pages.

We first use SPEC CPU2006[8] and STREAM[11] benchmarks to evaluate overheads and throughput of Hzmem when compression is disabled. Then we use datasets from Yelp Dataset Challenge[1] to measure effective memory increasing introduced by Hzmem when compression is enabled. Finally we conduct a series of benchmarks to test the new page fault overhead in order to show performance isolations guaranteed by Hzmem. We use consistent parameters for hot and cold pages detecting: watermark is 80% and detecting period is 10 seconds.

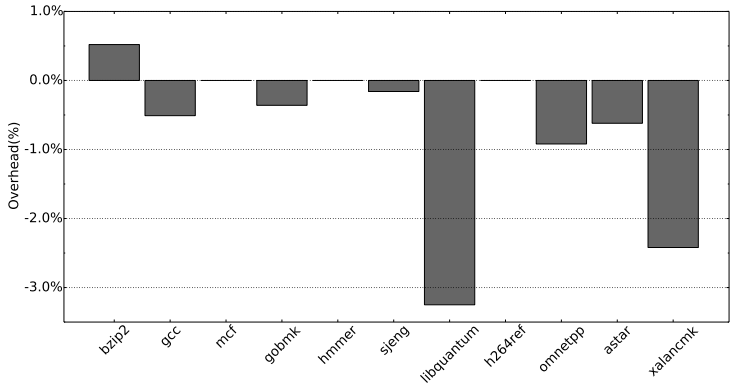


Fig. 3. Overhead of Hzmem relative to unmodified Linux

5.1 Overheads of Hzmem

Figure 3 shows the overheads introduced by Hzmem from SPEC CPU2006 benchmarks. To evaluate the overall overheads of Hzmem physical memory management, we utilize `hugectl` from `libhugetlbfs` to run the benchmarks. `libhugetlbfs` is a set of user tools making use of Linux `hugetlbfs`, which requires only re-linking binaries without modifications to source codes. `Hugectl` can remap the text and data segments of programs into memories backed up by

large pages and hook `libc` memory allocation functions `malloc` with mappings of large pages.

From the results, we can find that Hzmem slows down 3.25% in the worst case and 0.7% in average. The performance loss is mainly from the extra code path dealing with decompression/compression. Hzmem is not showing advantage but achieving comparative performance when memory is not under pressure.

5.2 Throughputs of Hzmem

Table 1. Throughput (MB/s) of benchmark `STREAM` when different sizes are applied.

	COPY		**SCALE**		**ADD**		**TRIAD**	
Size(m)	**Hzmem**	**Unmod**	**Hzmem**	**Unmod**	**Hzmem**	**Unmod**	**Hzmem**	**Unmod**
10	3035.7	3043.0	2594.3	2595.8	3244.0	3226.5	2930.7	2928.4
20	2318.0	2321.6	1970.0	1968.6	2298.3	2278.1	1839.0	1838.9
30	2315.5	2317.7	1973.2	1970.6	2395.9	2397.8	2214.7	2216.7
40	2319.0	2325.0	1972.2	1976.0	2294.5	2295.7	1825.9	1829.9
80	2503.4	2324.0	2131.7	1978.2	2440.6	2299.9	1956.0	1828.2
100	2383.6	2324.2	2029.5	1978.1	2327.5	2278.8	1875.9	1836.7

Table 1 shows the throughput of Hzmem using STREAM benchmark against unmodified Linux. STREAM is a synthetic memory bandwidth benchmark that measures the performance of four long vector operations: Copy, Scale, Add, and Triad. We configure STREAM to use different array sizes. From the results, we see that in small sizes from 10 million to 40 million the throughput is almost the same with the difference lower than 1%. On the contrary, with size from 80 million on, the difference gets larger. Hzmem has 7% larger throughput than unmodified Linux in the best case and 5% in average.

The reason is that Hzmem is based on our decoupled and flexible memory framework which uses new page descriptor for each large page without splitting or coalescing neighboring memory. When memory is under pressure and fragmented after long running, unmodified Linux tends to split or coalesce tremendous amounts of memory to meet large page memory allocation from user applications, thus bringing overheads and reducing throughput.

5.3 Effective Memory Increasing

By compressing large pages, Hzmem makes larger effective memory available to applications and avoids disk accesses or being killed due to `OOM killer`.

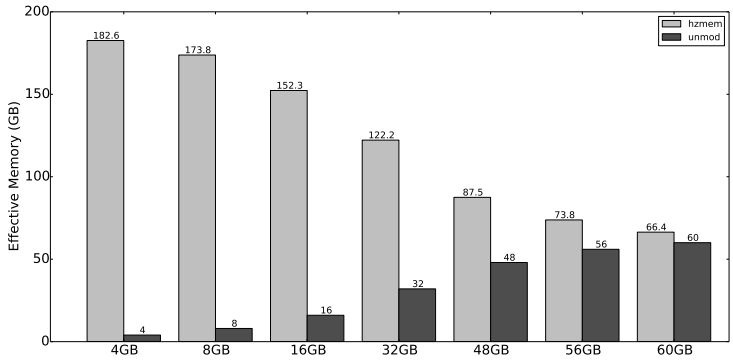


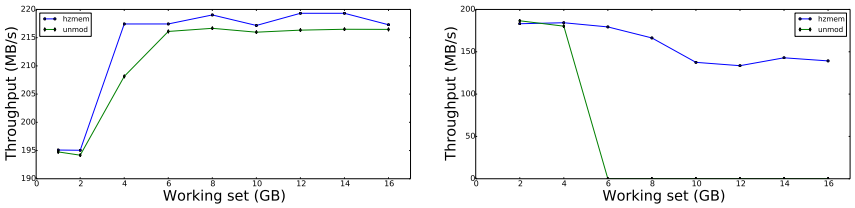
Fig. 4. Effective memory of Hzmem relative to unmodified Linux

Effective memory size increasing by compressing is dependent on the compression ratio. We measure an average compression ratio of 0.23 tested on dataset of Yelp Dataset using LZ4 algorithm. To evaluate the actual effective memory increasing introduced by Hzmem, we configure the same size of large memory in advance both in Hzmem and unmodified Linux. We run an application that keeps allocating memory using `hugetlbfs` API and writing memory until it fails. The largest memory it can obtain is the effective memory size.

Figure 4 shows effective memory increasing introduced by Hzmem against unmodified Linux using `hugetlbfs` API in different large memory size configurations in advance. We can see that unmodified Linux cannot increase effective memory size at all and the largest size available is the same as configured in advance. Hzmem can increase the effective memory size by 4465% but the gap is narrowing as the memory size configured in advance gets larger. This stems from the fact that Hzmem stores the compressed data in base page space. The smaller large page memory size is configured in advance the larger base page space can be used to store the compressed data from large page space. However, data compressing brings overheads and trade off between effective memory size and performance should be taken into consideration by system administrators.

5.4 Overhead of Page fault and Performance Isolation

Page faults in Hzmem involve data decompressing and lie in critical path of memory accesses. Thus performance of the new page fault handler of large pages is important. To evaluate the performance of the new page fault handling, we run an application that stresses heavily on the page faults. It first uses `hugetlbfs` API to allocate certain amounts of large pages and writes pages to trigger page faults. The amounts of large page to be allocated from user applications are divided into two groups dependent on whether beyond size of large pages configured in advance: non-overcommitted and overcommitted. To eliminate overheads of data compressing from page reclaiming, we configure size of large pages to be



(a) when not overcommitted (b) when overcommitted

Fig. 5. Throughput of page fault stress test

4 GB in advance and let the application sleep for 30s in the case of overcommitted and we subtract this period of time from running time for fairness when evaluating throughput. Another reason why we let the application sleep is that it can make the allocated pages as cold as possible, which stresses more heavily on the new page fault handler.

Figure 5(a) shows throughput of page faults stress test against unmodified Linux when not overcommitted. We can see that Hzmem has larger throughput than unmodified Linux but the difference is not larger than 9 MB/s, which means 4% in the best case.

Figure 5(b) shows throughput of page faults stress test against unmodified Linux when overcommitted. Since 4 GB of large pages are configured in advance, when allocating not over 4 GB the throughput are almost the same. When allocating over 4 GB large pages, the throughput of unmodified Linux becomes zero since it cannot increase effective memory size. However, when over 4 GB throughput of Hzmem are decreasing no more than 27%, which is apparently quite better than swapping involved in disk accesses which are much slower.

In the meantime, we also measure the CPU utilization caused by compressing daemons: 16.6% in the worst case and 11.0% in average. In our case, we have compressing daemons work on two nodes. If more daemons operate on more nodes, compressing overhead will be amortized and become much smaller on each CPU.

6 Conclusion

With trends towards running workloads that require big-memory, large page support and main memory compression are the techniques that developers often rely on to improve performance. However, applying the two techniques is beyond a matter of engineering. Combination of the two techniques is related to the low level design modifications and considerable effort. This motivates us to propose a completely new memory management framework that is decoupled and flexible for easy development and able to run simultaneously with the original memory manager. On top of the new memory management framework, Hzmem is large page memory management redesign with compression features independent from the base page memory management. It achieves competitive performance with

native large page supports, increases effective memory size and impacts little on other subsystems of operating system.

Acknowledgments. Many thanks to members of ARC Lab of Zhejiang University for their constructive comments and helps during the project. We would like to thank the anonymous reviewers for their feedbacks. This research is funded by National Key Technologies R&D Program of Ministry of Science and Technology of the People's Republic of China under Grant NO. 2016YFB0800201.

References

1. Yelp Dataset Challenge, https://www.yelp.com/dataset_challenge/
2. Basu, A., Gandhi, J., Chang, J., Hill, M.D., Swift, M.M.: Efficient virtual memory for big memory servers. Proceedings of the International Symposium on Computer Architecture pp. 237–248 (2013)
3. Clements, A.T., Kaashoek, M.F., Zeldovich, N.: RadixVM : Scalable address spaces for multithreaded applications. EuroSys pp. 211–224 (2013)
4. Corbet, J.: Huge pages part 1 (Introduction) (2010), https://lwn.net/Articles/374424/
5. Ekman, M., Stenstrom, P.: A robust main-memory compression scheme. Proceedings - International Symposium on Computer Architecture 00(C), 74–85 (2005)
6. Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A.D., Ailamaki, A., Falsafi, B.: Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In: ACM SIGPLAN Notices. vol. 47, pp. 37–48. ACM (2012)
7. Gerber, S., Zellweger, G., Achermann, R., Kourtis, K., Roscoe, T., Milojicic, D.: Not your parents ' physical address space. HotOS (2015)
8. Henning, J.L.: Spec cpu2006 benchmark descriptions. ACM SIGARCH Computer Architecture News 34(4), 1–17 (2006)
9. Huang, J., Qureshi, M.K., Schwan, K.: An Evolutionary Study of Linux Memory Management for Fun and Profit Methodology. Review-ATC'16 (2016)
10. Kwon, Y., Yu, H., Peter, S., Rossbach, C.J., Witchel, E.: Coordinated and Efficient Huge Page Management with Ingens. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) pp. 705–721 (2016)
11. McCalpin: STREAM benchmark (2002), http://www.cs.virginia.edu/stream/
12. Navarro, J., Iyer, S., Druschel, P., Cox, A.: Practical, transparent operating system support for superpages. ACM SIGOPS Operating Systems Review 36, 89 (2002)
13. Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Ongaro, D., Parulkar, G., Others: The case for RAMCloud. Communications of the ACM 54(7), 121–130 (2011)
14. Pekhimenko, G., Mowry, T.C., Mutlu, O.: Linearly compressed pages: a low-complexity, low-latency main memory compression framework. MICRO-46 Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) p. 489 (2013)
15. Tuduce, I.C., Gross, T.R.: Adaptive main memory compression. In: USENIX Annual Technical Conference, General Track. pp. 237–250 (2005)
16. Zaitsev, P., Tkachenko, V.: Evaluating Database Compression Methods: Update (2016), https://www.percona.com/blog/2016/04/13/evaluating-database-compression-methods-update/