Thinking about A New Mechanism for Huge Page Management

Xinyu Li[1], Lei Liu ✉[1], Shengjie Yang[1], Lu Peng[2], Jiefan Qiu[3]
[1]Sys-Inventor Lab, SKLCA, ICT, CAS; [2]LSU; [3]ZJUT

ABSTRACT

The Huge page mechanism is proposed to reduce the TLB misses and benefit the overall system performance. On the system with large memory capacity, using huge pages is an ideal choice to alleviate the virtual-to-physical address translation overheads. However, using huge pages might incur expensive memory compaction operations due to memory fragmentation problem, and lead to memory bloating as many huge pages are often underutilized in practice.

In order to address these problems, in this paper, we propose SysMon-H, a sampling module in OS kernel, which is able to obtain the huge page utilization in a low overhead for both cloud and desktop applications. Furthermore, we propose H-Policy, a huge page management policy, which splits the underutilized huge pages to mitigate the memory bloating or promotes the base 4KB pages to huge pages for reducing the TLB misses based on the information provided by SysMon-H. In our prototype, SysMon-H and H-Policy work cooperatively in OS kernel.

1 INTRODUCTION

We are in the era of big data and cloud computing. In this era, applications have rapidly increasing memory footprints and demand for throughput than ever before [20,22,27]. For example, the widely used cloud workloads, e.g., Memcached and Redis, have several hundred GB/TB-level memory demands on YouTube, Facebook and Twitter's data center [2,6]. Meanwhile, designing an efficient memory management mechanism for computer systems with a large memory capacity is always challenging the existing Operating System (OS) [7,10,14,24].

Chasing a high overall system performance, previous studies [8,10,24] propose schemes on using huge page for cloud computing environments. Huge page helps to reduce the number of TLB misses and thus benefits the overall system performance [10,14,16]. Modern architecture has TLB entries for huge pages. For example, The Intel Nehalem, Sandy Bridge/Skylake series processors now have 512/1536 TLB entries for huge pages [9]. Yet, the huge page is not always a free lunch, and the OS is now wrestling with the following challenges. (1) In reality, the huge page is often with low utilization (the term "utilization" stands for the fraction of a memory page that is actually used for data storage), incurring memory bloating and thereby wasting memory. Previous efforts [2, 16] show, in Redis, using huge page wastes 69% memory compared with using only base 4KB page. (2) Allocating huge pages often incur significant overheads on memory compaction. As system ages, physical memory is fragmented, thus OS has to compact physical memory to create contiguous regions for huge page allocations. In many cases, as some kernel-level pages cannot be moved, OS fails to have contiguous regions for allocating huge pages [11,12,24]. Even the compaction is successful, the overheads are ineligible [12]. Due to these problems, some reports [16,24] claim using huge page may incur performance degradation and recommend disabling the huge page mechanism.

To make the huge page actually useful, the efforts in [8, 23,24] propose the adaptive huge page allocation approaches, which are application transparent and support multiple page sizes (i.e., enabling small page or huge page accordingly). Now, Linux uses the Transparent Huge Pages (THP) mechanism, which allocates a 2MB huge page for every memory request and enables compaction operation if there is no contiguity memory space for the huge page allocation. However, compaction routine often fails and brings nothing benefits but overheads. Going with frequent latency spiking brought by compactions [1–3], THP also cannot avoid the low utilization problem for huge pages. And, memory bloating caused by enabling huge page may waste a large amount of memory, leading to swapping data with the hard disk, especially in the cases, where memory resource is not ample.

APSys '19, August 19–20, 2019, Hangzhou, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6893-3/19/08…$15.00
https://doi.org/10.1145/3343737.3343745

✉ Lei Liu is the corresponding author (lei.liu@zoho.com; liulei2010@ict.ac.cn). This project is founded in Sys-Inventor Lab led by Lei Liu. Part of the technique in this paper is based on Lei Liu's previous work "Going Vertical in Memory Management" [19] in ISCA-2014 (the corresponding author is Lei Liu).

In this paper, ***we believe that making OS aware of huge page utilization is necessary for cloud environments.*** For example, database workloads often perform poorly with huge pages, because they tend to have sparse rather than contiguous memory access patterns. If OS always allocates huge pages for such sort of workloads, the memory utilization will become very low and might lead to expensive swapping operations. To our knowledge, even the studies have the idea of using multiple page sizes according to the memory system's current status and workloads patterns, e.g. the efforts in [8], ***it is still a fundamental problem for OS to understand the huge page utilization, and thereby adapting memory policies accordingly becomes difficult in reality.***

Towards this end, we design a practical OS-level monitoring tool (SysMon-H), which is capable of capturing the huge page utilization with low overhead at runtime for cloud workloads. SysMon-H is the first work that combines TLB monitoring and page-table walks to achieve both high accuracy and low overhead for monitoring huge pages. And, we propose H-Policy as a new memory policy for huge page management in OS. The sampling results from SysMon-H can be used by H-Policy to enable appropriate memory management policies, e.g., splitting huge pages into small ones or promoting base pages to huge pages. H-Policy and SysMon-H work cooperatively in OS kernel.

2 BACKGROUNDS AND MOTIVATION

Linux kernel supports using huge pages since the version 2.6 (e.g., Debian since 2.6.32)[1]. There are two ways of enabling the huge pages in Linux: (1) using hugetlbfs [1, 4], people can reserve a large number of memory pages with a consecutive physical address for huge page allocations. However, this approach is not flexible, as the reserved pages can only be used for huge page allocations; (2) THP can transparently allocate huge pages without the human involvements. On a system with THP, when a page fault occurs, THP tries to find a block with 512 contiguous physical pages (2MB) in buddy system [5,11]. However, as system ages, there will be lots of fragments in memory space, thus THP has to enable the time-consuming memory compactions to create a huge page (2MB contiguous pages) [23]. In extreme cases, where the compaction operation fails due to the unmovable pages, THP has to merely return a basic 4KB page with a long latency. The latest work in [24] skips the hybrid page blocks (those blocks with unmovable pages) during the compactions, therefore reducing the allocation latency. [16] proposes an asynchronous allocation to create contiguous memory spaces for huge pages, thus reducing the overhead brought by compactions.

In addition to prior work, we have the following insights into huge page management. (1) It will be necessary to make OS aware of the huge page utilization at runtime, as memory bloating always wastes a large amount of memory. In the cases where memory is limited, OS needs to split the huge pages with lower utilization into small pages, and then other applications can use them. The existing work does not discuss this topic in details. (2) OS should have a flexible memory mechanism and adapt the appropriate policy according to the workloads' patterns and memory system's status at runtime. In this work, we try to answer two questions: ***can we have a new memory policy for huge page management? How to make OS aware of the huge page utilization in a low overhead?***

3 MONITORING THE HUGE PAGE UTILIZATION

To address the above-mentioned questions, the first step is to design a practical OS kernel-level module for capturing the memory pages' utilization (including both of the Huge 2MB and base 4KB pages). We use Linux with the kernel version 3.16 in our experiments, and our experiment platform is with an Intel Nehalem i7-2.8GHz CPU (with 512 TLB entries for Huge pages) and 32GB main memory.

3.1 SysMon-H

Many previous studies [17,19,21,25,26] periodically check the access_bit in PTEs (Page Table Entry) to monitor the temperature (i.e., access frequency) of the memory pages. However, with the increasing of memory footprint, frequently checking the access_bit is not cost-effective. For example, in our experiment with Redis, the sampling overhead achieves 6 seconds in a specific sampling window when memory footprint is around 20GB, affecting the user experience in practice. Obviously, we need a new approach for monitoring the cloud workloads.

In this paper, we design SysMon-H, an OS module (enhanced from [13,19,28]) to collect the number of TLB misses for huge pages instead of merely relying on access_bit. The core idea of SysMon-H is from the observations that the cold pages (rarely accessed pages) have only a small number of TLB misses; In contrast, hot pages usually incur a large number of TLB misses, as they are frequently required to be loaded into TLB. For a progr-

[1] Linux now supports 2MB and 1GB huge pages. In this paper, we use 2MB huge pages in our experiments. The term huge page refers to 2MB huge page.

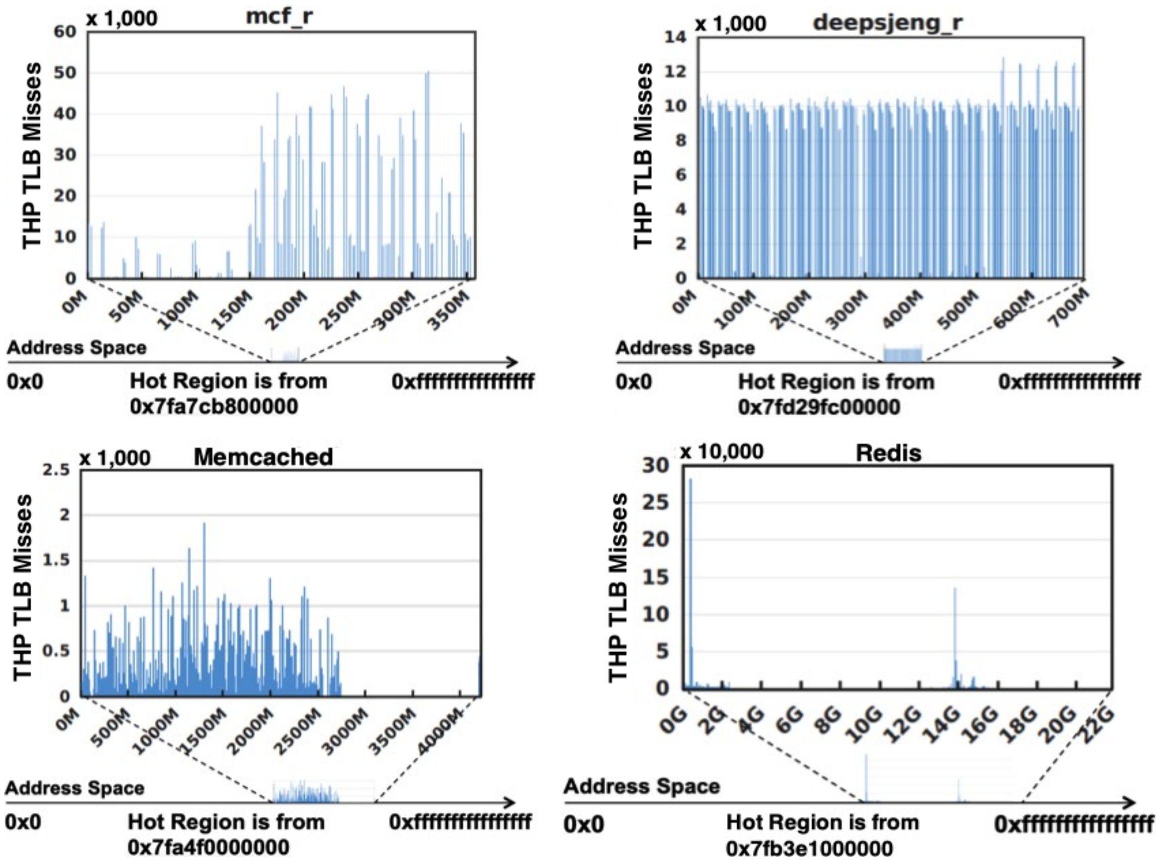

Figure 1: The number of TLB misses of Huge Pages for 4 applications in a 5-seconds sampling window.

-am with a large working set, hot pages will be swapped in/out of the TLB repeatedly, thus by monitoring the number of TLB misses, SysMon-H can obtain the distribution of memory accesses for the huge pages in the entire memory address space. Usually, a high number of TLB misses stand for a huge page with a high temperature ("hot" huge page) and been touched frequently; in contrast, a huge page with a low temperature might be considered split into base pages (4KB pages) to mitigate the memory bloating problem. Our approach is reasonable, because TLB has a limited number of entries and it employs a hash function based replacement algorithm to prevent the side-channel attack from TLB, thus each page's TLB entry has the almost equal opportunity to be swapped out [15].

Previous work [13] can merely obtain the overall number of TLB misses for a specific application. In our design, by attaching a shadow array in application's VMA, SysMon-H can obtain the distributions of TLB misses in the application's address space at runtime, and can have the per huge page-level TLB misses consequently. To our knowledge, SysMon-H is the first work that can get such kind of the information at OS level.

We show the sampling results using SysMon-H for Memcached, Redis, deepsjeng_r and mcf_r in SPEC CPU 2017. Figure 1 shows the results of the number of TLB misses for the 4 applications in their address spaces, in a 5 seconds sampling window. SysMon-H can capture the hot regions that consist of the frequent accessed huge pages whose TLB misses are high, and find out the cold regions in which the huge pages are with a low number of TLB misses, i.e., underutilized huge pages. For example, the Memcached's memory accesses spans a 4GB address space, and most of the hot huge pages are mainly in the left half of the address space; the distribution of hot huge pages of Redis is uneven among its 22GB address space; mcf_r's memory accesses are in a 350MB range, and the result of deepsjeng_r shows most of the pages are accessed without a significant difference. Based on the SysMon-H's sampling results, huge pages are ranked according to their TLB misses. The pages with a low number of TLB miss are considered underutilized ones. SysMon-H records this information at runtime.

However, only considering the TLB misses might not be enough. People may intuitively assume that some "very hot"

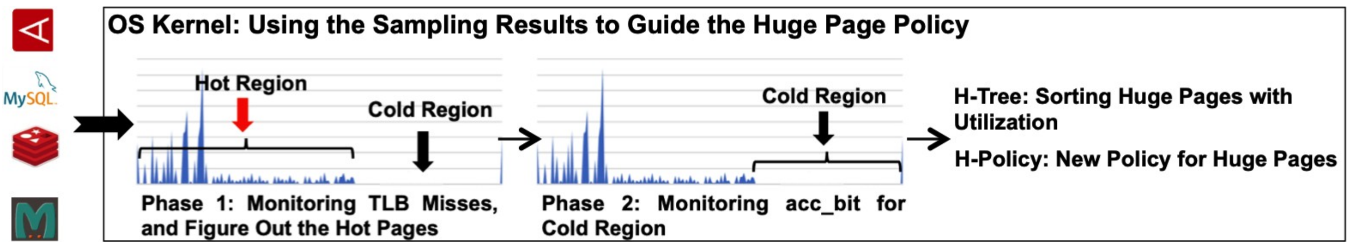
OS Kernel: Using the Sampling Results to Guide the Huge Page Policy
Hot Region
Cold Region
Phase 1: Monitoring TLB Misses, and Figure Out the Hot Pages
Cold Region
Phase 2: Monitoring acc_bit for Cold Region
H-Tree: Sorting Huge Pages with Utilization
H-Policy: New Policy for Huge Pages

Figure 2: Workflow of SysMon-H.

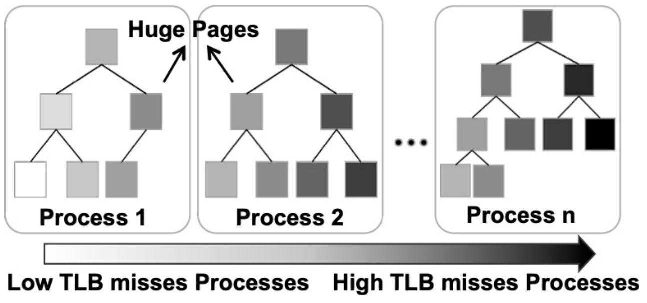

Figure 3: H-Tree for Each Application.

pages might be kept in the TLB, and thus cause fewer TLB misses as cold pages do. We conduct experiments to show how many pages are in this category. In our experiments, the pages whose TLB miss count is below 10 in the sampling period (5s) are classified as cold pages, and we further check their access_bits to discern whether they are hot pages or not. Here "hot" refers to the pages that are touched in 3 consecutive scan intervals (1s). The experimental results show that pages belong to this category exist, but the number of them is not high. With respect to our benchmarks, this special category of pages accounts for below 0.44% of the pages that are classified as cold ones in general. The preliminary results are basically consistent with the claim in [15]. More experimental results will be reported in future extension articles.

Moreover, besides the huge pages, SysMon-H can also collect the similar information for the base 4KB pages. More design details are in following sections.

3.2 The Design Details of SysMon-H

In this section, we show more design details of SysMon-H and its interaction with other OS components. Figure 2 shows the overall idea of SysMon-H. It has two phases. In Phase 1, SysMon-H captures the access patterns by monitoring the number of TLB misses primarily, and further checks the access_bit for those cold regions to find out these "very hot" huge pages (phase 2). The rest of the pages are those with low utilization. SysMon-H works as an OS module for long-running servers, and periodically performs sampling for every 30 seconds (collect information for 5 seconds) in our prototype.

We design the prototype of SysMon-H based on the Linux kernel with version 3.16. SysMon-H collects the information for each application one by one. For a specific application, it collects the TLB misses for its pages for 5 seconds, and then finds out the hot and cold regions; For these cold regions, SysMon-H further checks the access_bit in these huge pages' PTE for three times, each of which is with a 1-second interval. If the access_bit is 1 (i.e., touched) for two times, SysMon-H will mark the page as hot, otherwise, it will be marked as a cold one. SysMon-H monitors the huge pages and the base 4KB pages.

For a specific application, SysMon-H constructs an H-Tree for every application, and sorts its huge pages (denoted by Page_Struct) in H-Tree according to the number of TLB misses. The core data structure of H-Tree is the binary sort tree, and each application has its own H-Tree. The huge pages with more TLB misses (and the pages are classified as "very hot" in phase 2) will be considered the high utilized ones. In our prototype, the watermark high_tlb is defined as 50, indicating that for a specific page whose number of TLB misses achieve 50 will be considered a page with high utilization.

Illustrated in Figure 3, we show how the outputs from SysMon-H are organized. Using the H-Tree, with the watermark high_tlb as the root, pages are organized according to their access frequency for each application. The underutilized pages are located in the left part of the tree, thus OS can easily find these huge pages with low overheads. H-Tree has its application's total number of TLB misses during a specific sampling interval (i.e., 30 seconds in our prototype). If the memory bloating is serious, our approach will select the application whose H-Tree has the least number of the TLB misses and then starts to split its underutilized huge pages (i.e., the nodes in the left part of its tree). More details are in section 4. Note that the constants in our design (e.g., watermark, sampling interval, and etc.) are empirical values based on the analyses of programs from cloud and SPEC CPU 2017 applications.

These values can be adjusted as necessary in the conditions of environmental changes.

3.3 The Sampling Overheads

We show the overheads brought by SysMon-H. As its sampling routine has two phases, we conduct a two-stepped experiment to show the advantage of it.

Step-1: We compare the overheads caused by SysMon-H (mainly relies on monitoring TLB misses) to the approach that only samples the access_bit [21,26] for the workloads Redis and mcf_r in SPEC CPU 2017. Our experimental results show that sampling access_bits (i.e., employing 200 loops during a sampling window) for mcf_s several hundred huge pages merely brings 0.19% overheads. In contrast, monitoring TLB misses for mcf_r brings 0.14% overheads in our experiment, which is lower than only monitoring access_bit.

As mcf_r in SPEC CPU 2017 has a below 1GB memory footprint, accounting for at most 500 2MB huge pages, the sampling overheads of the two approaches are not significant.

In contrast, for the cloud workload, which has a large number of huge pages, and often exhibits the random-like and irregular memory access patterns, monitoring TLB misses for it will not incur unnecessary overheads such as monitoring access_bit does (even for finding out a small number of randomly touched hot pages, monitoring access_bit has to frequently scan the entire address space). In the case of Redis with 22GB memory footprint on server side, our experimental results show below 0.1% sampling overheads caused by monitoring TLB misses on the Redis server (i.e., the client will not have obvious latency when it access the Redis server), while in contrast, only sampling access_bit brings 2.3% overheads, which may affect the user experience. Redis has the largest memory footprint in our experiments (illustrated in Figure 1), thereby we think the experimental results could be representative.

Step-2: We compare the overheads brought by handling the corner cases. As illustrated in Figure 2, SysMon-H needs to find out these "very hot" pages that are with a low number of TLB misses in its second phase from those pages that were once classified as cold. As SysMon-H knows these pages are primarily cold ones, it only checks them for 3 times (with 1-second intervals). Nevertheless, the prior approaches [21,26] do not have such knowledge and have to frequently check every huge page for capturing the access frequency, even for these are cold ones. Tracking every huge page in address space in this way without any distinctions brings significant overheads in reality, especially for the cloud computing workloads, whose memory footprints are very high. Our experimental results show that SysMon-H reduces this sampling overhead to around 1/20 in Step-2.

4 THE ART OF H-POLICY DESIGN

SysMon-H works as a kernel module in OS to obtain the memory pages' utilization, guiding the memory management mechanism (i.e., H-Policy). In this section, we introduce H-Policy, which leverages the knowledge provided by SysMon-H and manages the huge pages accordingly. The H-Policy design has the following key rules.

(i) H-Policy allows splitting the huge pages that are with low utilization into base 4KB pages according to the current memory status. H-Policy uses a watermark, i.e., high_pressure (90%), to denote the amount of allocated memory in the system. When the allocated memory amount achieves 90%, H-Policy starts to aid the memory bloating by splitting the huge pages that are with low utilization. At the first step, it chooses the application that has the lowest overall TLB misses (i.e., the application that has a relatively lower number of TLB misses); then, for the application, H-Policy splits the huge pages with the low number of the TLB misses (i.e., the huge pages are with low utilization) by referring to the application's H-Tree. After splitting a huge page, the freed base pages (marked by explicit hints) return to buddy system. As mentioned before, all of the required information is in this tree in Figure 3.

(ii) H-Policy will promote the base 4KB pages that have a high number of TLB misses in a consecutive 2MB memory space to a huge page. As mentioned before, hot pages often cause a high number of TLB misses. H-Policy has another watermark, i.e., promote_space (90%), indicating that 90% of the 4KB pages in a specific 2MB space are with a high number of TLB misses, and H-Policy should promote this 2MB space to a huge page. In our design, H-Policy promotes pages in every 30 seconds. During this process, H-Policy migrates pages and compacts data for creating a consecutive 2MB physical space with a linear mapping to the 2MB virtual space. H-Policy uses the OS page migration and data compaction primitives in Linux kernel. More details can be found in [1,4].

(iii) Preserving order-9 slab in buddy system for huge page allocations. H-Policy is design based on the Linux' buddy system [5,18], which has 11 free page lists organized by slabs with orders ranging from 0 to 10, and each list with an order R organizes pages in blocks that have 2^R continuous 4KB physical pages. Upon a memory allocation request, one larger block with a higher order can be split into smaller blocks of lower orders when there are not sufficient free pages in lower order slabs. For example, an order-9 block that has 512 (2^9) 4KB pages can be split into 10 blocks with 256 (order 8), 128, 64, 32, 16, 8, 4, 2 pages and two blocks with only one page, respectively. The split blocks are linked to slabs with smaller orders. In reality, large blocks are split quickly, and thus OS has lots of external memory fragments. In such cases, allocating huge pages (2MB with 512 4KB pages) becomes either time consuming due to memory compaction or impossible. In our design, H-Policy does not split the blocks in the order-9 slab for 4KB page allocations unless pages in other order slabs are all allocated (i.e., delaying splitting the blocks that can be directly used for allocating huge pages). Doing in this way, H-Policy can potentially have more 2MB blocks for huge page allocations in 2^9 free list, and OS can allocate a huge page in constant time without other unnecessary operations, e.g., breaking the large blocks or data compactions. This approach can also prevent the 2MB blocks in OS from being split quickly.

(iv) H-Policy will aggressively create large physical memory blocks by merging the adjacent small page blocks, in the cases where OS has a large number of fragments. H-Policy tries to migrate fewer pages for creating the large blocks, as enabling page migration in Linux kernel may incur performance slowdown. In our design, H-Policy tracks the "holes" in physical memory by using counters in the buddy system's slabs and is able to have large blocks by removing the relatively small holes in memory address space via the page migration operations (large holes bring more page migration overheads). Moreover, H-Policy has the interface to tune its performance.

Note that the watermark and other parameters in our prototype can be modified according to specific needs. In our design, H-Policy is orthogonal with the existing buddy system in Linux kernel.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we show SysMon-H, an OS kernel-level monitoring module, which can capture the utilization of huge pages by combining both TLB monitoring and PTE sampling. With the help of SysMon-H, OS is able to make better use of huge pages. Our experimental results show that SysMon-H works well and brings low overheads. Furthermore, we propose H-Policy, a new memory policy for huge page management. With H-Policy, OS splits the underutilized huge pages for mitigating the memory bloating and promotes base pages to huge pages for performance, adaptively. And, as system ages, H-Policy can potentially have fewer memory fragments than original Linux kernel as it preserves 2MB contiguous blocks in buddy system for huge page allocations. SysMon-H and H-Policy work together in our design.

Our future work includes: (1) developing an efficient memory compaction approach and a page migration mechanism for reducing the data migration overheads, and therefore can further improve the overall system performance; (2) designing a dedicated memory framework to efficiently supports a certain type of workload; (3) deploying H-Policy on platforms using NVM, and merging new NVM techniques together [29,30]; (4) besides the memory management mechanism, we would like to study the impact on operating system's core components, e.g., FS and I/O [31-33], on emerging systems with large memory capacity. We hope our work could provide a valuable reference for future related studies.

ACKNOWLEDGEMENT

We thank the reviewers for their valuable comments. This project is supported by the National Key Research and Development Program of China under Grant No. 2017YFB1001602 and the NSF of China under Grant No. 61502452. Xinyu Li and Shengjie Yang are student members in Sys-Inventor Lab supervised by Lei Liu.

REFERENCES

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

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

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

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

[5] http://en.wikipedia.org/wiki/Buddy_memory_allocation.

[6] https://en.wikipedia.org/wiki/Memcached.

[7] N. Agarwal and Thomas F. Wenisch, Thermostat: Application transparent page management for two-tiered main memory. In ASPLOS, 2017.

[8] R. Ausavarungnirun, et al, Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In Micro, 2017.

[9] A. Awad, et al, Avoiding TLB shootdowns through self-invalidating TLB entries. In PACT, 2017.

[10] A. Basu, et al, Efficient virtual memory for big memory servers. In ISCA, 2013.

[11] D. P. Bovet and M. Cesati, Understanding The Linux Kernel. O'Reilly Media, Inc. 2005.

[12] J. Corbet, Memory compaction. https://lwn.net/Articles/368869/.

[13] J. Gandhi, et al, BadgerTrap: a tool to instrument x86-64 TLB misses. In ACM SIGARCH Computer Architecture News (CAN), 2014.

[14] J. Gandhi, et al, Efficient memory virtualization: Reducing dimensionality of nested page walks. In Micro, 2014.

[15] B. Gras, et al, Translation Leak-aside Buffer: Defeating Cache Sidechannel Protections with TLB Attacks. In USENIX Security, 2018.

[16] Y. Kwon, et al, Coordinated and efficient huge page management with ingens. In OSDI, 2016.

[17] S. Lee, et al, CLOCK-DWF: A writehistory-aware page replacement algorithm for hybrid PCM and DRAM memory architectures. In IEEE TC, 2014.

[18] L. Liu, et al, A software memory partition approach for eliminating bank-level interference in multicore systems. In PACT, 2012.

[19] L. Liu, et al, Going Vertical in Memory Management: Handling Multiplicity by Multi-policy. In ISCA, 2014.

[20] L. Liu, et al, Memos: A full hierarchy hybrid memory management framework. In ICCD, 2016.

[21] L. Liu, et al, Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random? In IEEE TC, 2016.

[22] O. Mutlu, More than Moore Technologies for Next Generation Computer Design. Springer, Chapter Main Memory Scaling: Challenges and Solution Directions, 2015

[23] J. Navarro, et al, Practical, transparent operating system support for superpages. In OSDI, 2002.

[24] A. Panwar, et al, Making Huge Pages Actually Useful. In ASPLOS, 2018.

[25] M. Xie, et al, SysMon: Monitoring Memory Behaviors via OS Approach. In APPT, 2017.

[26] X. Zhang, et al, Towards practical page coloring-based multicore cache management. In EuroSys, 2009.

[27] Y. Zhang, et al, Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In ASPLOS, 2015.

[28] H. Zhao, et al, Bandwidth and Locality Aware Task-stealing for Manycore Architectures with Bandwidth-Asymmetric Memory. In ACM TACO, 2018.

[29] L. Liu, et al, Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems. In IEEE TPDS, 2019.

[30] S. Chen, et al, Efficient GPU NVMRAM Persistence with Helper Wraps. In ACM/IEEE DAC, 2019.

[31] H. Liu, et al, HMFS: A hybrid in-memory file system with version consistency. In JPDC, 2018.

[32] F. Lv, et al, Dynamic I/O-aware scheduling for batch-mode applications on chip multiprocessor systems of cluster platforms. In JCST, 2014.

[33] F. Lv, et al, WiseThrottling: a new asynchronous task scheduler for mitigating I/O bottleneck in large-scale datacenter servers. In J. of Supercomputing, 2014.