SmartMD: A High Performance Deduplication Engine with Mixed Pages

Fan Guo[1], Yongkun Li[1,2], Yinlong Xu[1,3], Song Jiang[4], John C. S. Lui[5]

[1]*University of Science and Technology of China*

[2]*Collaborative Innovation Center of High Performance Computing, NUDT*

[3]*Anhui Province Key Laboratory of High Performance Computing, USTC*

[4]*University of Texas, Arlington*

[5]*The Chinese University of Hong Kong*

Abstract

In hypervisor-based virtualization environments, translation lookaside buffers (TLBs) misses may induce two-dimensional page table walks, which may incur a long access latency, and this issue becomes worse with ever increasing memory capacity. To reduce the overhead of TLB misses, large pages (e.g., 2M-pages) are widely supported in modern hardware platforms to reduce the number of page table entries. However, memory management with large pages can be inefficient in deduplication, leading to low utilization of memory, which is a precious resource for a variety of applications.

To simultaneously enjoy benefits of high performance by accessing memory with large pages (e.g., 2M-pages) and high deduplication rate by managing memory with base pages (e.g., 4K-pages), we propose Smart Memory Deduplciation, or SmartMD in short, which is an adaptive and efficient management scheme for mixed-page memory. Specifically, we propose two lightweight schemes to accurately monitor pages' access frequency and repetition rate, and present a dynamic and adaptive conversion scheme to selectively split or reconstruct large pages. We implement a prototype system and conduct extensive experiments with various workloads. Experiment results show that SmartMD can simultaneously achieve high access performance similar to systems using large pages, and achieves a deduplication rate similar to that applying aggressive deduplication scheme (i.e., KSM) at the same time on base pages.

1 Introduction

In modern computers, processors use page tables to translate virtual addresses to physical addresses. To accelerate the translation, TLB was introduced to cache virtual-to-physical address mappings. While TLB is critical to system's performance, its misses carry high penalty for accessing the page table in memory. In particular, in a system employing hypervisor-based virtualization, the hypervisor and guests maintain separate page tables, and TLB misses will lead to high-latency two-dimensional page table walks. Previous works [13, 17] show that this is often the primary contributor to the performance difference between virtualized and bare-metal systems. In fact, the overhead of TLB misses has become one of the primary bottlenecks of memory access.

Moreover, while memory size becomes increasingly larger, TLB's capacity cannot grow at the same rate as DRAM. To reduce TLB miss ratio, large pages are introduced in many modern hardware platforms to reduce the number of page table entries required to cover a large memory space. For example, the X86 platform supports 2M-page and 1G-page, while the ARM platform supports 1M-page and 16M-page [9].

It is important to note that different VMs on the same host machine often run similar operating systems (OSes) or applications. It is highly likely that there exists a great deal of redundant data among different VMs [14]. Thus, we can save memory space by removing redundant data and sharing only a single copy of the data among different VMs (also known as memory deduplication). However, for memory systems with large pages (e.g., 2M-pages), our experiments show that it is hard to find duplicate large pages even the memory contains a large amount of redundant data. In other words, deduplication in unit of the large page is ineffective and usually saves only a small amount of memory space.

To enable more effective deduplication, current OSes exploit an aggressive deduplication approach (ADA), which aggressively splits large pages (e.g., 2M-pages) to base pages (e.g., 4K-pages) and performs deduplication among base pages [22]. However, after the splitting, the memory space covered by translation entries in the TLB can be significantly reduced. Although ADA saves more memory space, accessing the split large pages significantly increases TLB miss ratio and degrades access performance. Moreover, the reconstruction of split large

pages is not well supported in current OSes. In a system that keeps running, there are increasingly more split pages, leading to continuous degradation of memory access performance.

In this paper, our objective is to maximize memory saving with deduplication while keeping high memory access performance on a server hosting multiple VMs. In particular, we propose SmartMD, to maximize memory saving while keeping high performance of memory access. The main idea is to split cold large pages with high repetition rate to save memory space, and at the same time, to reconstruct split large pages when they become hot to improve memory access performance. The key challenges are how to efficiently monitor repetition rate and access frequency of pages, and how to dynamically conduct conversions between large pages and base pages so as to achieve both high deduplication rate and high memory access performance. The main contributions of this work can be summarized as follows.

- We propose two lightweight schemes to monitor pages on their access frequency and repetition rate. In particular, we introduce counting bloom filters and sampling into the monitor such that it can accurately track pages' status with very low overhead. Additionally, we propose a labeling method to identify duplicate pages during the monitoring, which can greatly accelerate the deduplication process.
- We propose an adaptive conversion scheme which selectively splits large pages to base pages, and also selectively reconstructs split large pages according to the access frequency and repetition rate of these pages and memory utilization. With this bidirectional conversion, we can take both benefits of high memory access performance with large pages and high deduplication rate with base pages.
- We implement a reconstruction facility by selectively gathering scattered subpages of a split large page, and then carefully re-create descriptor and page table entry of the split large page. As a result, the memory access performance can be improved by reconstructing split large pages which turn hot.
- We implement a prototype and conduct extensive experiments to show the efficiency of SmartMD. Results show that SmartMD can simultaneously achieve high memory access performance similar to that of large page-based systems, and high deduplication rate similar to that produced by aggressive deduplication schemes, such as KSM.

The rest of the paper is organized as follows. We introduce the background of memory virtualization, large page management, and current aggressive deduplication technology in Section 2. We motivate the design for improving the aggressive deduplication policy in Section 3. In Section 4, we discuss the design of various techniques used in SmartMD. In Section 5, we describe the experiment setup and present the evaluation results to show the efficiency of SmartMD. Section 6 discusses the related work and Section 7 concludes the paper.

2 Background

2.1 Memory Virtualization

To efficiently utilize limited memory space, a high-performance server hosting virtual machines (VMs) usually dynamically allocates its memory pages to VMs on demand. Because of the dynamic allocation, physical addresses of the memory pages allocated to a VM are often not contiguous. So in a hypervisor-based virtualized system, a VM uses guest's virtual addresses (GVA) and guest's physical addresses (GPA) for its memory access. GPA are logical addresses on the host and they are mapped to host physical addresses (HPA). To improve the address translation performance from GPA to HPA, extended page tables (named by Intel) or nested page tables (named by AMD) [12] have been introduced. With the extended page tables [1], a VM will carry out a two-dimensional page walk to access its data with two steps. First, a GVA is translated to its corresponding GPA using guest's page tables. Second, the GPA is further translated to its corresponding HPA using extended page tables.

When using base pages (i.e., 4KB pages in X86-64 system), both the guest's page table and extended page table are composed of four levels. Accessing each level of the guest's page table will trigger the traversal of the extended page table. In the worst case, a two-dimensional page walk will require 24 memory references [12, 23], which is apparently unacceptable. A commonly practice to accelerate the address translation is to cache frequently used global mapping from GVA to HPA in the TLB [27].

However, when the memory becomes increasingly large the page tables consistently grow, and as a result the hit ratio of TLB reduces. To increase the hit ratio of TLB and further speedup the address translation in a system with a large amount of memory, large pages have been widely adopted in today's systems.

2.2 Advantage of Using Large Pages

A large page is composed of a fixed number of contiguous base pages. For example, in a X86-64 system, OS

[1] In this paper we will use Intel's extended page tables as an example, though the design and conclusions derived from the evaluation results are also applicable to the system using nested page tables.

Benchmark	Host: Base Guest: Large	Host: Large Guest: Base	Host: Large Guest: Large
SPECjbb	1.06	1.12	1.30
Graph500	1.26	1.34	1.68
Liblinear	1.13	1.14	1.37
Sysbench	1.07	1.09	1.20
Biobench	1.02	1.18	1.37

Table 1: Benchmark performance with selective use of large pages. Details of the benchmark programs are described in Section 5. The performance is normalized against the one for running the benchmark on the system using base pages in both guest and host OSes.

uses one 2MB-page entry to cover a contiguous 2MB region of memory for its address translation, instead of using 512 4KB-page entries to cover it. In a virtual environment, large pages can be supported in both guest's page tables and extended page tables [12]. With large pages, the page table becomes significantly smaller, and much larger memory space can be covered by a TLB table of the same size. In this way, using large pages helps to increase TLB hit ratio for global mappings. In particular, it reduces maximum number of memory references in a 2D page walk after a TLB miss from 24 to 15 [12].

To show improvement of memory access performance with large pages, we run experiments with various benchmarks in a KVM virtual machine. Detailed configuration of the virtual machine is described in Section 5, and we present the experimental results in Table 1. We can see that the performance can be significantly improved for most of the benchmarks even if we use large pages only in guest's OS or in host's OS. In particular, if we use large pages in both OSes, the performance of Graph500 is improved by 68% over the baseline system in which only base page is used.

2.3 Impact of Using Large Pages on Memory Deduplication

Usually there is a great deal of duplicated data residing in the memory of a virtualized machine [14]. Deduplication among different VMs will lower the memory demand and keep memory from being overcommitted. However, even though there can be plenty of duplicate data in the memory, there can be very few duplicate large pages. While the deduplication removes duplicate data at the granularity of page, it may not be effective with the use of large pages. Our experiments show that ADA can save 13.7% - 47.9% of used memory for the selected benchmarks, but deduplication in unit of large page saves only 0.8% - 5.9% of used memory (see Table 2). That is, deduplication among different VMs in unit of large page can save very little memory. Thus, major OSes support an aggressive deduplication approach (ADA), which splits a large page into base pages and then applies deduplication on base pages [22], such as KSM in Linux.

Policy	Benchmark	Memory Saving	Performance
Large Page w/o ADA	Graph500	0.37 GB(3.4%)	1
	SPECjbb2005	0.40 GB(5.9%)	1
	Liblinear	0.32 GB(2.0%)	1
	Sysbench	0.09 GB(0.8%)	1
	Biobench	0.20 GB(1.4%)	1
Large Page with ADA	Graph500	5.18 GB(47.9%)	0.695
	Specjbb2005	1.83GB(26.9%)	0.922
	Liblinear	3.79 GB (23.7%)	0.846
	Sysbench	2.83 GB(18.0%)	0.867
	Biobench	1.88 GB(13.7%)	0.910

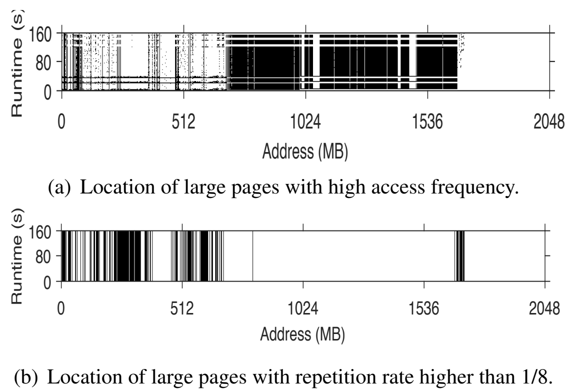
Table 2: Memory saving and performance of large-page-based systems with/without running ADA (aggressive deduplication approach), which splits all large pages. When ADA is not used, deduplication is applied at the large-page granularity. Memory saving is normalized against the memory demand in the system without using any deduplication. The performance is normalized against that for the system using large pages without applying ADA.

3 Motivations

3.1 Aggressive Deduplication

When a large page contains duplicate subpages, ADA will split it into base pages and then perform deduplication on these base pages. Although ADA has the potential to save significant amount of memory space, the page tables become much larger, which will reduce the hit ratio of TLB and increase memory access time. The worst scenario of ADA is that it splits a large page that has high access frequency and low repetition rate (or the percentage of duplicate subpages belonging to the large page). And it compromises memory access performance and produces little memory saving.

We carry out experiments to show the statistics of memory pages. Fig. 1 shows the distributions of large pages with high access frequency or high repetition rate of a VM running SPECjbb. From Fig. 1(a), we can see that the SPECjbb benchmark constantly accesses some large pages throughout its entire run time while other large pages are rarely accessed. Fig. 1(b) shows that majority of large pages with high repetition rate appears only in few memory regions. Comparing Fig. 1(a) with Fig. 1(b), we find that many large pages have high access frequency but few duplicate subpages. In the meantime, there exist large pages with many duplicate subpages and low access frequency. In short, with ADA that selects large pages for splitting without considering page access frequency and repetition rate, the benefit of its limited memory saving can be more than offset by the degraded memory access performance for many applications.



(a) Location of large pages with high access frequency.

(b) Location of large pages with repetition rate higher than 1/8.

Figure 1: Memory usage of SPECjbb.

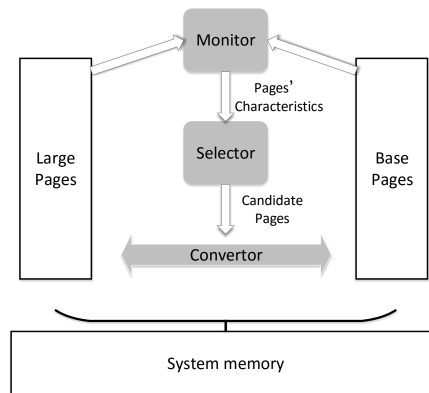


Figure 2: Illustration of SmartMD's Architecture.

We experimentally compare memory saving and memory access performance in the system using large page and ADA. The results are shown in Table 2. With ADA, the system saves 13.7%-47.9% of memory space but is slowed down by up to 30.5% due to increased TLB misses after splitting large pages. Specifically, the percentage of large pages drops to 16% on average. On the other hand, retaining large pages preserves high memory access performance, but it loses opportunities of reducing memory usage. Thus, current memory management scheme is inadequate for virtualized systems running memory-intensive applications.

To this end, we propose SmartMD, a selective deduplication scheme that assigns each large page a priority of being split for potential deduplication according to its access frequency and repetition rate. SmartMD splits large pages with high repetition rate and low access frequency and performs deduplication among their subpages to save memory while maintaining high access performance.

3.2 Difficulties of Converting Base Pages to Large Pages

Major OSes support splitting of large pages to produce more deduplication opportunities. However, the reconstruction of base pages back into large pages is not well supported [22, 8]. In particular, only large pages whose base pages are not shared with those in other large pages can be reconstructed. Furthermore, the reconstruction may substantially compromise system performance. Meanwhile, instead of releasing free pages back to the host, a VM often keeps these pages for its incoming applications. Thus, the conversion of base pages to large pages in current OSes may cause incremental degradation of memory access performance. In this work, we propose an approach to efficiently reconstruct large pages to improve memory access performance.

3.3 The Challenges

Monitoring pages' statuses. SmartMD needs to track pages' access frequency and repetition rate, which are not directly disclosed by current OSes. Meanwhile, monitoring these parameters will introduce additional overheads. Thus, we need to design an efficient monitoring mechanism with low overhead.

Choosing right pages. Splitting large pages into base pages and reconstructing base pages into large pages may have big negative impacts on memory access performance. SmartMD must carefully select right pages to split and reconstruct for maximal efficacy and minimal side effect. Furthermore, applications' demands on memory and CPU may change dynamically, so SmartMD needs to identify current performance bottleneck and resource constraint and to provide an adaptive conversion mechanism between large pages and base pages to alleviate the situation.

Reconstructing large pages. SmartMD provides an approach to reconstruct base pages into large pages. However, implementation of the approach can be challenging, because splitting a large page not only changes its descriptor and page table entries of its subpages, but also breaks the contiguity of its subpages. Even worse, some subpages might have been freed after splitting, which imposes great difficulty on the reconstruction process.

4 Design of SmartMD

In this section we will overview the design of SmartMD followed with design details on each of its components.

4.1 Overview of SmartMD

As shown in Fig. 2, SmartMD is composed of three modules, *Monitor*, *Selector*, and *Converter*. In the Monitor, we provide two lightweight schemes to track number of

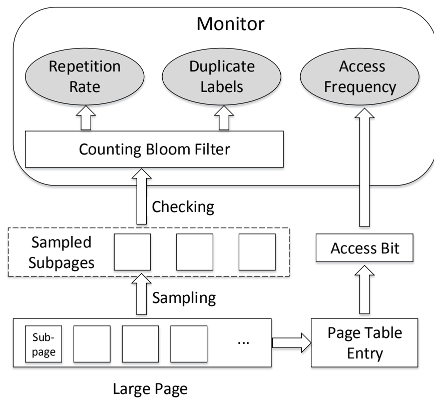


Figure 3: Design of the Monitor.

duplicate subpages, or the repetition rate for large pages. This information will be used by the Selector to select large pages for splitting or base pages for reconstruction. In particular, we propose an algorithm which dynamically performs the selection according to the current memory utilization, data access frequency, and large-page repetition rate. Finally, the Converter performs the conversion between large pages and base pages.

4.2 The Monitor

The Monitor uses a thread to periodically scan pages to measure memory utilization as well as page access frequency and repetition rate. Fig. 3 illustrates the techniques used in the Monitor and its workflow.

Monitoring memory utilization and page access frequency. We note that the OS already provides a utility to monitor and disclose the size of free memory space in a system. However, it does not provide a utility to directly monitor and disclose page access frequency. To address this issue, SmartMD periodically scans access bit of each page to gauge pages' access frequency. It clears the access bits of all pages at the beginning of a monitoring period, and checks each of them after $check_interval$ seconds. If the access bit of a page is one, which is set due to a reference to the page in the period, SmartMD will increment its access frequency by one. Otherwise, the page was not accessed in the last period and its access frequency is decremented by one. If a large page has been split, we check the frequencies of its subpages and see if any of them is larger than zero. If yes, we increment frequency of the the original large page by one. However, we keep the frequency value always in the range from 0 to N, where N is a positive integer, and will not change it beyond the range. We initialize a page's frequency to $N/2$ when the system starts.

Detecting repetition rate of pages. To measure the repetition rate of a large page (or the percentage of duplicate base pages in the large page), existing approaches use comparison trees to identify duplicate pages [11]. However, they carry high CPU overhead. In contrast, SmartMD uses a counting bloom filter for an approximate identification.

The counting bloom filter is a one-dimensional vector, and each of its entries is a 3-bit counter. As shown in Fig. 4, when scanning a large page, SmartMD uses the counting bloom filter to check whether its subpages are duplicates or not. Specifically, when checking a subpage, SmartMD applies three hash functions on the subpage's content to calculate the indexes of its corresponding counters. For each subpage, SmartMD also records its signature, which is produced by applying a hash function on its content and is used to represent the page. If a page is checked for the first time (i.e., its recorded signature is not found), SmartMD will increase its corresponding counters by one. Otherwise, if all of the counters are greater than one, we consider this page as a duplicate one. If a page is modified, SmartMD decrements each of its current counters by one and increments each of its new counters by one. In addition, if a page is released, SmartMD also decrements each of its counters by one.

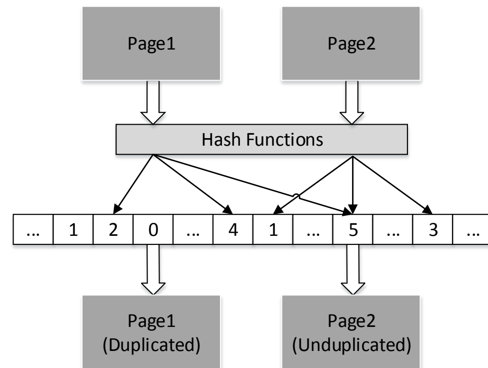


Figure 4: Identification of duplicate pages by using a counting bloom filter.

To make a trade-off between memory overhead and identification accuracy, SmartMD sets the size of the counting bloom filter, in terms of counters in it, as eight times of the number of base pages in the system. With this configuration, SmartMD can ensure that the false positive of the bloom filter is less than 3.06% [1].

SmartMD adopts a sampling-based approach to further accelerate the identification. Specifically, the Monitor first samples some subpages in a large page and calculates their hash values. It then checks whether these sampled subpages have been modified during the previous monitoring time period by comparing their current signatures with the ones on record. If a large page has been modified or is scanned for the first time during the sampling process, the Monitor will scan all the subpages

to update their signatures and insert them into the counting bloom filter. Meanwhile, SmartMD calculates the repetition rate of the large page. Otherwise, the Monitor calculates the repetition rate only among the sampled subpages, instead of all subpages in the large page, so as to reduce the overhead. For the subpages identified by the Monitor as duplicates, SmartMD labels them as a hint to the deduplication component to improve its efficiency. Specifically, when a large page is being split, SmartMD uses KSM to deduplicate redundant pages. KSM searches the labeled pages in the comparison trees to speed up the deduplication process. SmartMD organizes each large page's metadata about its access frequency and repetition rate in a linked list.

Our sampling-based detecting algorithm can help to substantially reduce the CPU overhead for most workloads. Experiments show that the ratio of misidentification of duplicate pages is less than 5% by sampling only 25% subpages in a large page. In particular, the counting bloom filter improves the efficiency of SmartMD on three aspects. First, it helps SmartMD to obtain approximate repetition rate of large pages with a small overhead. By using the repetition rate, we can avoid splitting large pages with low repetition rate. Second, it labels identified duplicate pages to speed up the deduplication process of SmartMD. Third, it reduces the number of nodes in the deduplicaiton trees by only splitting large pages with high repetition rate.

4.3 The Selector

To improve memory access performance, the Selector chooses candidate large pages for splitting based on two metrics, namely access frequency and repetition rate.

Identifying cold and hot pages. Upon knowing pages' access frequency from the Monitor module, the Selector divides all pages into three categories, cold, warm, and hot, with two thresholds, $Thres_{cold}$ and $Thres_{hot}$. If a large page's frequency value is smaller than $Thres_{cold}$, it is designated as cold. If its frequency value is greater than $Thres_{hot}$, it is a hot page. All other pages are designated as warm. We denote the gap between the two thresholds $(Thres_{hot} - Thres_{cold})$ as $length_{warm}$. Note that the state of warm is a transition one between the cold and hot states. We introduce it to avoid switching between the hot and cold states too often.

Identifying duplicate pages. We set a repetition rate threshold, $Thres_{repet}$, for the Selector to select candidate pages. In particular, the Selector only selects large pages whose percentages of duplicate subpages are more than $Thres_{repet}$ for splitting, and we name these pages as *duplicate large pages* or simply *duplicate pages*. It is important to set $Thres_{repet}$ properly so as to obtain a high deduplication rate with minimal number of split large pages. In our experiments, we find that by setting $Thres_{repet} = 1/8$, SmartMD can deduplicate more than 95% of duplicate subpages and split 40% fewer pages than traditional aggressive deduplication approach.

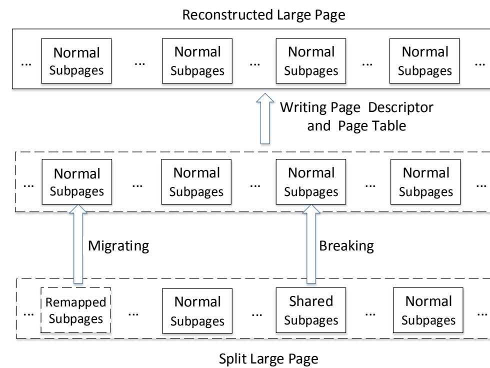


Figure 5: Process of reconstructing split large pages.

Selector Workflow. When scanning a large page, the Selector first reads its access frequency. If this page has been designated as cold, the Selector will further determine whether its repetition rate is greater than $Thres_{repet}$. If yes, this page is ready for splitting. On the other hand, when selecting split large pages for reconstruction, the Selector chooses only hot pages as candidates.

4.4 The Converter

The converter is responsible for the conversion between large pages and base pages, including the splitting of large pages and the reconstruction of split pages. The splitting process can be realized by calling a system API, while OSes do not well support the reconstruction functionality. We implement a utility in SmartMD to reconstruct split large pages. Fig. 5 illustrates this process, which consists of the following three steps.

(1) **Gathering subpages.** To reconstruct a split large page, we need to ensure that all of its subpages currently reside in memory and are not deduplicated with other pages. If some subpages have been deduplicated, we generate a duplicate copy for each of these subpages, and migrate all subpages to a contiguous memory space before reconstructing.

(2) **Writing page descriptor.** Once all subpages of a split large page have been gathered, we re-create page descriptor of the large page from the page descriptors of all subpages.

(3) **Writing page table.** We use a single page entry to map the reconstructed large page, and invalidate old entries about the original subpages.

As the cost of gathering subpages for reconstruction of large pages can be high, we propose two gathering mechanisms to reduce the number of subpages that have to be migrated. Specifically, if most of the subpages of a large page still stay in their original physical memory locations, we conduct *in-place gathering*, in which we migrate the subpages that have been relocated back to their original memory locations after migrating pages currently occupying the locations elsewhere. Otherwise, if most subpages of a split large page have been relocated from their original memory locations, we conduct *out-of-place gathering*, in which a contiguous memory space of the size of a large page is allocated and all of the large page's subpages are migrated into the space. Because of existence of spatial locality in the memory access, it is expected that for a particular workload either a high percentage of subpages of a split large page stay in the original locations or a high percentage of them do not. Our experiments show that for most benchmarks we tested, the percentages are larger than 90%. By adaptively applying the gathering mechanisms, we can significantly reduce gathering cost and the reconstruction overhead.

Adaptive page conversion. To reduce the cost of conversion between large pages and base pages, we develop an adaptive conversion scheme to improve performance of SmartMD based on the ratio of allocated memory size to total memory size, i.e., utilization of the memory space. The idea is that if the system has sufficient free memory space, we use only large pages for high memory access performance. On the other hand, if memory utilization becomes high and memory page swapping may occur, we split large pages into base pages for a high deduplication rate. Specifically, the adaptive page conversion scheme uses four parameters to guide its conversion decision, including two thresholds about memory utilization (mem_{low} and mem_{high}) and two thresholds about access frequency ($Thres_{cold}$ and $Thres_{hot}$). In each monitoring period, we first check the memory utilization, and then tune the parameter $Thres_{cold}$ accordingly so as to dynamically identify pages to be split. In particular, if the memory utilization is less than mem_{low}, we decrement $Thres_{cold}$ by one to make more pages stay in the warm or hot states and keep them from being split for high memory access performance. If the memory utilization is greater than mem_{high}, indicating that memory is in high demand, we increase $Thres_{cold}$ by one to allow more large pages to be considered as cold pages and be eligible for being split so as to achieve higher deduplication rate. Similar to a page's frequency value, we also keep $Thres_{cold}$ in the range from 0 to N, where N is a positive integer, in the process.

5 Evaluation

To show its efficacy and efficiency, we implement a SmartMD's prototype on Linux 3.4 and conduct experiments using QEMU to manage KVM. Our experiments run on a server with two Intel Xeon E5-2650 v4 2.20GHz processors, 64GB RAM, and a 2TB WD hard disk (WD20EFRX). Both the host and guest OSes are Ubuntu 14.04. We boot up four VMs in parallel, each of which is assigned one VCPU and 4GB RAM, and all VMs are hosted on one physical CPU. In our experiments, we focus only on 2MB and 4KB pages, which are commonly used in most applications. We run the following benchmark programs in each VM. Their memory demands without deduplication are listed in Table 3.

- **Graph500 [2].** Graph500 generates and compresses large graphs. It also runs breadth-first search on the graph. We run Graph500 in each guest VM with the same scale (22) and edgefactor (16). We generate graphs initialized differently to ensure that graphs in different VMs are different. We use average number of edges traversed in a VM per second as the performance metric of the benchmark.
- **SPECjbb [6].** SPECjbb is a benchmark for evaluating performance of server-side Java business applications. We run SPECjbb in each VM and use the average bops (business operations per second) of all VMs as its performance metric.
- **Libliner [5].** Libliner is a suite of linear classifiers for a data set with millions of instances and features. We run SVM, one benchmark program in Liblinear, on the urlcombined dataset. The performance metric is average execution time of the program running in different VMs.
- **Sysbench [7].** Sysbench is a multi-threaded benchmark for database. We run sysbench on Mysql by storing all data in the buffer pool of Mysql. We use the average number of queries performed by a VM per second as the performance metric.
- **Biobench [10].** Biobench is a suite of bioinformatics applications. We run Mummer, a program in Biobench on the human-chromosomes dataset [4], and measure its average execution time in different VMs.

Graph-500	SPECjbb-2005	Lib-linear	Sys-bench	Bio-bench
2.7GB	1.7GB	4.0GB	2.93GB	3.42GB

Table 3: Memory usage of each VM w/o deduplication.

We compare SmartMD with three other schemes on both performance and memory usage. The first one is

KSM, which uses the aggressive deduplication approach to split all large pages to achieve the best deduplication rate. The second one is named *no-splitting*, which preserves all large pages and performs deduplication in unit of large page to achieve the best access performance. The third one is Ingens [22], which is the state-of-the-art scheme using mixed pages to make a trade-off between access performance and memory saving. Default values of the parameters used in the experiments are listed in Table 4. We adopt the same rate at which for the schemes to scan and identify duplicate pages for a fair comparison.

Parameter	Value	Description
$monitor_period$	6s	scanning period of the monitoring thread
$check_interval$	2.6s	interv. of checking access bits
$Thres_{repet}$	1/8	thresh. of repetition rate
mem_{high}	90%	thresh. of high mem. utilization
mem_{low}	80%	thresh. of low mem. utilization
$page_to_scan$	1024	number of pages scanned by dedup-thread in each scan
$sleep_millisecs$	20ms	time to sleep after each scan of the dedup-thread

Table 4: Default parameter setting.

Note that with the adaptive page conversion scheme described in Section 4.4, large page will not be split for deduplication if there is a sufficient amount of free memory. In the evaluation of SmartMD on its effectiveness and efficiency (see Section 5.1∼5.4), we use fixed nonzero $Thresh_{cold}$ and $Thresh_{hot}$, instead of the adaptive conversion scheme, to make sure that SmartMD comes into effect even when the server has abundant free memory. Specifically, we set the range of a page's access frequency from 0 to 4. Meanwhile, instead of allowing $Thresh_{cold}$ to be decremented to 0 due to low memory utilization, we fix it at 1 so that large pages eligible for splitting may still be produced even if the system has enough free memory. In addition, we set $Thresh_{hot}$ to 3. We set initial access frequency of each page to 2, lying between $Thresh_{cold}$ and $Thresh_{hot}$, to ensure that it has a chance to be classified as either hot or cold page.

To evaluate effectiveness of the adaptive conversion scheme, we run experiments with SmartMD in a memory-constrained system (Section 5.5). In particular, we limit the host's memory space by running an in-memory file system (hugetlbfs [3]) to occupy a certain amount of memory space on the host. Pages held by hugetlbfs cannot be deduplicated or swapped out. In this way, we can flexibly adjust size of the host's memory available for running benchmark programs.

5.1 Overhead of SmartMD

CPU overhead. We first run Graph500 to compare the CPU overhead of SmartMD with the other two memory deduplication schemes, KSM and Ingens. The results are shown in Table 5. Both the monitoring thread and deduplication thread use additional CPU cycles. KSM uses aggressive deduplication without tracking status of the pages. However, without knowing whether a large page contains duplicate subpage(s), it has to scan all large pages and in each large page determines whether each of its subpages is a duplicate, leading to high CPU overhead in its deduplication. As shown in Table 5, KSM spends more CPU time than Ingens and SmartMD by 26% and 34%, respectively. SmartMD takes more CPU time on monitoring each large's access frequency and repetition rate. In contrast, Ingens monitors only access frequency. Accordingly, the monitoring thread of SmartMD induces 7.8% higher CPU overhead than that of Ingens. With the knowledge on access frequency and repetition rate of each large page, as well as on which of its subpage are duplicates, SmartMD can more efficiently and precisely locate large pages for effective deduplication. As shown Table 5, SmartMD's deduplication thread spends 9.4% lower CPU time than that of Ingens. Comparison of deduplication effectiveness with Ingens will be presented in Section 5.3.

	Monitor thread	Dedup thread	Total
KSM	0	33.5%	33.5%
Ingens	5.3%	21.3%	26.6%
SmartMD	13.1%	11.9%	25.0%

Table 5: Average CPU utilization sampled in every second.

Memory overhead. SmartMD uses 3 bits to store each of the eight counting bloom filters for each base page. Since the size of a base page is 4KB, the ratio of extra memory space used to store the filters is only $(3bits \times 8) \div (4KB) = 3/2^{12}$. For each large page, we use 32B to store its access frequency and repetition rate, as well as some necessary pointers. Since the size of a large page is 2MB, SmartMD requires additional $32B \div 2MB = 1/2^{16}$ of the memory space for large pages. For example, 16 GB memory is used during the running of Libliner on four VMs. SmartMD needs about 12MB to store bloom filers for base pages and 0.25MB to store metadata for large pages. Apparently the memory overhead of SmartMD is negligible.

5.2 Performance and Memory saving

In this section, we compare SmartMD with two commonly used mechanisms in major OSes, which are KSM or no-splitting, using different benchmark programs on their performance and memory usage. Comparison with Ingens will be presented in Section 5.3. By aggressively splitting any large pages to maximize deduplication opportunities, KSM can achieve the highest memory sav-

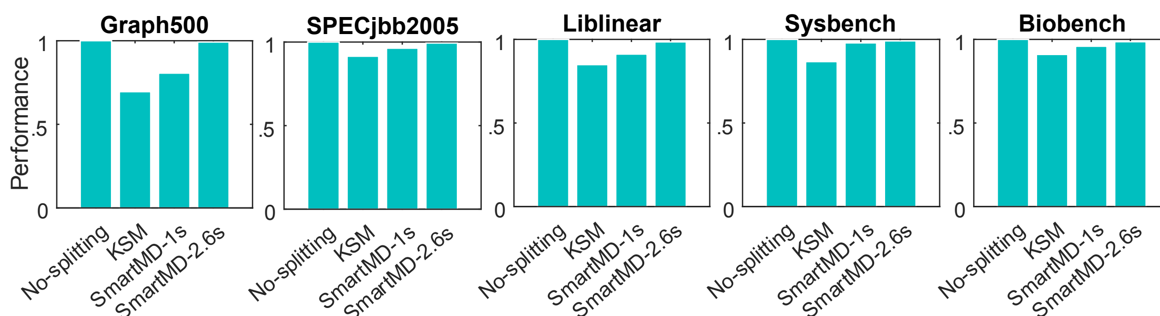


Figure 6: Performance of the benchmarks under various deduplication policies.

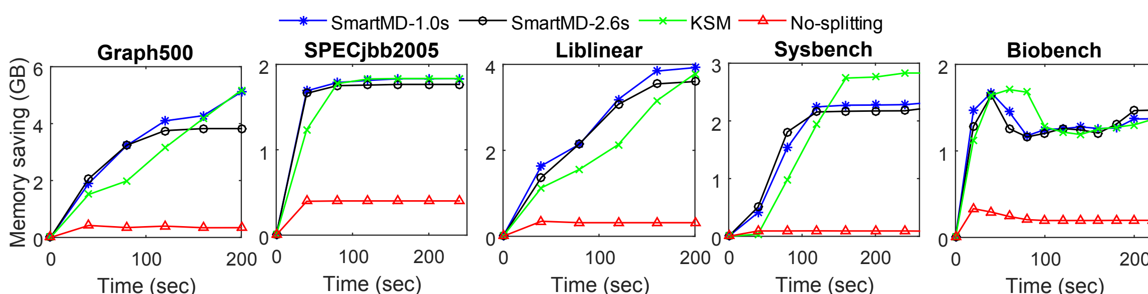


Figure 7: Memory saving under various deduplication policies.

ing. On the other hand, no-splitting represents an optimization only on performance by preserving all large pages. Here we study the trade-off made by SmartMD between performance and memory saving by comparing it with the KSM and no-splitting schemes.

We first show performance of the benchmarks by using SmartMD, KSM and no-splitting in Fig. 6, where we normalize the performance, whose metrics are introduced in the description of the benchmarks in Section 5, against that of the no-splitting scheme. In the experiments, we use two different check_interval values (1.0s and 2.6s) in SmartMD to vary the time period between reseting access bits and its next reaching of the bits. Accordingly, SmartMD is named SmartMD-1s and SmartMD-2.6s, respectively. Fig. 6 shows that for the benchmarks SmartMD achieves nearly the same performance as no-splitting by using a larger check_interval. In contrast, SmartMD improves KSM's performance by up to 42.7% by only spliting necessary large pages.

Experiment results on memory saving are shown in Fig. 7. Because no-splitting does not perform splitting of large pages and conducts deduplication in the unit of large page, it reduces memory usage by a small percentage (6% or less). In contrast, SmartMD and KSM can reduce memory usage by a much larger amount, which is usually 4x to 31x as large as the saving received in no-splitting. Fig. 7 also show , we can also see that in general SmartMD reduces about the same amount of memory as KSM. Interestingly, in some execution periods of some benchmarks, such as Liblinear, SmartMD reduces more memory than KSM. By using counting Bloom filters and labeling of duplicate pages, SmartMD can complete its scan of memory to find duplicate pages much faster than KSM, and carry out deduplication in a more timely manner. For example, to reduce memory usage of Liblinear by 3.2GB SmartMD-2.6s and KSM take 118s and 161s, respectively.

Looking into Figs. 6 and 7, we can see that SmartMD takes both benefits on memory saving and access performance. Specifically, SmartMD can save 4x to 21x as much memory as the no-splitting scheme while keeping similar access performance. For example, with Graph500 SmartMD can save 3.82 GB memory space, or 35.4% of the total memory, which is 9x the memory space saved by no-splitting. In the meantime, SmartMD can achieve up to 15.8% of performance improvement over KSM while achieving a memory saving similar to KSM.

Additionally, SmartMD can be configured to tune the weight of its optimization goals between access performance and memory saving. With SmartMD, we can improve either the access performance or memory saving while minimally compromising the other goal. For example, the performance of Sysbench is improved by 12.9% with increasing checking interval from 1.0s to 2.6s. Meanwhile, the memory saving only decreases by 4.3%. This is because SmartMD splits only large pages with low access frequency and high repetition rate. In this way, SmartMD can ensure that each splitting can bring benefit of memory saving but incur small negative impact on memory access performance. Furthermore, base pages can be opportunistically converted back to large pages to benefit the performance of SmartMD.

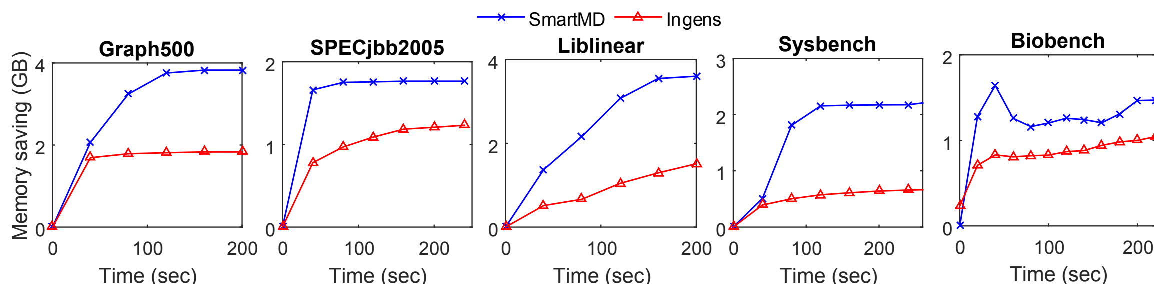


Figure 8: Comparison of memory saving between Ingens and SmartMD.

	Ingens	SmartMD
Graph500	0.989	0.992
SPECjbb2005	0.991	0.994
Liblinear	0.987	0.991
Sysbench	0.982	0.989
Biobench	0.976	0.982

Table 6: Performance normalized to that of No-splitting.

	Single-CPU	NUMA
Graph500	0.8%	1.6%
SPECjbb2005	0.6%	2.1%
Liblinear	0.9%	1.8%
Sysbench	1.1%	2.6%
Biobench	1.8%	3.9%

Table 7: Performance degradation by using SmartMD on NUMA. The degradation is calculated against the performance of No-splitting with the same benchmark.

5.3 Comparison with Ingens

Kwon et al. [22] proposed Ingens to enable conversion from base pages to large pages to maintain high memory access. It also selectively splits large pages for more effective deduplication. However, in the selection of large pages, it only considers access frequency and does not take into account of repetition rate. In addition, it does not consider page access frequency in the decision of reconstruction of large pages. Table 6 shows the performance of the benchmark programs with SmartMD and Ingens. For SmartMD check_interval is set at 2.6s. Fig. 8 shows the memory saving of SmartMD and Ingens. We see that SmartMD can save 1.3x to 3.5x as much memory as Ingens while still keeping performance of SmartMD to that of Ingens. While Ingens splits any large pages that are considered cold, it has to throttle generation of cold pages to keep memory access performance close to that of no-splitting. This is achieved by postponement of checking accessing bits. However, this approaches leaves fewer pages available for deduplication. SmartMD can more precisely identify the right large pages (with low access frequency and high repetion rate) for splitting. It is less likely to conduct unnecessary splitting. SmartMD also performs necessary reconstruction of large pages to keep high memory performance.

5.4 Performance in a NUMA Environment

In the above experiments, all VMs are hosted on one physical CPU in a NUMA system. However, if they are hosted on different CPUs, deduplication may make accesses of originally local pages become more expensive ones of remote pages, causing performance degradation.

To study performance impact of the NUMA architecture, we place two VMs on one physical CPU, and another two on a different CPU and re-run the benchmarks with SmartMD. The performance results are shown in Table 7. As shown, running SmartMD in the NUMA environment does cause larger performance degradation. However, the NUMA impact is very small, as SmartMD only splits large pages into base pages and deduplicate them only for those with low access frequency. Thus, even if many pages are deduplicated and relocated, only a very limited number of remote accesses are induced.

5.5 Performance in a Memory Overcommitted System

In this section, we evaluate the performance with different memory loads: no-overcommitted, slight-overcommitted and severe-overcommitted, which correspond to scenarios where the ratios of memory demand of an application to the usable memory size as 0.8, 1.1, and 1.4, respectively. We compare the performance of benchmarks using KSM, Ingens, and No-splitting, and the performance results are shown in Fig. 9. We can see that when the system has sufficient memory, performance of SmartMD is close to that of No-splitting. This is because when the memory utilization is low, SmartMD sets the cold threshold ($Thresh_{cold}$) to zero to keep large pages from being split.

With the increase of the host's memory load, the access performance of No-splitting drops much faster than other three schemes. With less effective deduplication, No-splitting has a larger memory demand. When the demands is larger than usable memory size, it will cause

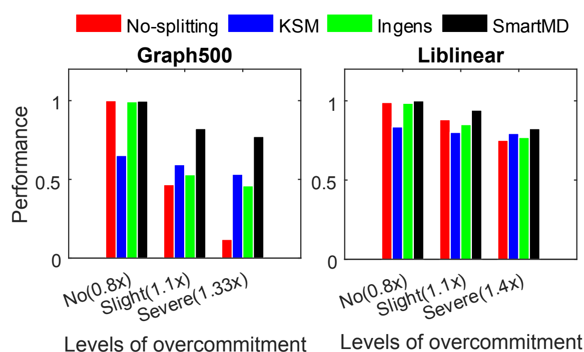


Figure 9: Performance in overcommitted systems.

more serious swapping of the program's working set between the memory and the disk, significantly slowing down the program's execution. With few pages deduplicated and larger memory demand than SmartMD, Ingens also shows significantly degraded performance in a memory overcommitted system.

SmartMD outperforms the other three schemes in the memory overcommitted systems. For example, for Graph500 SmartMD achieves up to 38.6% of performance improvement over other schemes. Using intelligently selective and adaptive conversion between large pages and base pages, SmartMD can make a better tradeoff between memory saving and access performance under different levels of memory overcommitments.

6 Related Work

Management of large pages. To efficiently use large pages, researchers proposed schemes to manage pages of different sizes [16, 26]. For example, Navarro et al. [25] provide a tool for FreeBSD to support multiple page sizes with contiguity-awareness and fragmentation reduction. Gorman et al. [18] propose a placement policy for physical page allocator, which mitigates fragmentation and increases contiguity by grouping pages according to whether the pages can be migrated. Their subsequent work [19] proposes an API for applications to explicitly request huge pages. Different from SmartMD, the above works do not consider memory deduplication.

Memory Deduplication. Memory deduplication has attracted attention of many researchers [11, 24, 20, 29, 28, 15, 21, 30]. In-memory deduplication technique was first implemented in VMWares ESX server [30], which requires no assistance from guest OSes and performs transparently in the hypervisor layer. KSM [11] is implemented as a kernel thread, which periodically scans memory pages to detect duplicate pages. Miller et al. [24] find that data in the page cache are more likely to be duplicates. They propose a memory deduplication scanner named XLH to identify duplicate pages. Gupta et al. propose Difference Engine [20] to deduplicate partial base pages with partial redundancy. The above works can be considered as aggressive deduplication schemes whose sole objective is to reduce memory usage. However, they do not consider impact of using large pages on deduplication efficacy as well as performance impact of splitting large pages.

The Ingens Deduplication Ingens [22] is a recently proposed memory deduplication scheme most similar to SmartMD. Ingens provides a coordinated transparent huge page support for the OS and hypervisor. In contrast, SmartMD achieves higher memory saving while maintaining similar access performance with its three advantages. (1) SmartMD selectively splits large pages according to their access frequency and repetition rate, while Ingens only considers pages' access frequency. (2) SmartMD reconstructs split large pages based on their access frequency, while Ingens reconstructs a large page as long as most of its subpages are utilized. (3) SmartMD adaptively selects pages for splitting and reconstruction, and uses sampling-based counting bloom filters and duplication labels to reduce overhead.

7 Conclusion

In this work, we propose SmartMD, an adaptive and efficient scheme, to manage memory with pages of different sizes. SmartMD can simultaneously take both the benefit of high performance by accessing memory with large pages, and the benefit of high deduplication rate by managing memory with base pages. Experimental results show that compared to KSM and no-splitting, SmartMD can either saves more memory space with similar memory access performance, or achieves higher memory access performance with similar memory saving.

8 Acknowledgements

We thank the anonymous reviewers and our shepherd, Don Porter, for their valuable comments and suggestions. In the work, Yongkun Li was partially supported by Anhui Provincial Natural Science Foundation (1508085SQF214). Yinlong Xu was partially supported by National Natural Science Foundation of China (61379038). Song Jiang was partially supported by US National Science Foundation (1527076). John C. S. Lui was partially supported by Hong Kong General Research Fund (14208816). Yongkun Li is the corresponding author.

References

[1] bloomfilter. `https://en.wikipedia.org/wiki/Bloom_filter`.

[2] Graph500. `http://www.graph500.org/specifications`.

[3] Hugetlbfs. `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`.

[4] Human chromosomes. `http://mummer.sourceforge.net/applications.html`.

[5] Liblinear. `https://www.csie.ntu.edu.tw/~cjlin/liblinear/`.

[6] SPECjbb2005. `https://www.spec.org/jbb2005/`.

[7] Sysbench. `https://github.com/akopytov/sysbench`.

[8] The big khugepaged redesign. `https://lwn.net/Articles/634384/`.

[9] Page sizes among architectures. `https://en.wikipedia.org/wiki/Page_(computer_memory)`, 2017.

[10] ALBAYRAKTAROGLU, K., JALEEL, A., WU, X., FRANKLIN, M., JACOB, B., TSENG, C.-W., AND YEUNG, D. Biobench: A benchmark suite of bioinformatics applications. In *ISPASS* (2005), IEEE.

[11] ARCANGELI, A., EIDUS, I., AND WRIGHT, C. Increasing memory density by using ksm. In *Proceedings of the linux symposium* (2009), Citeseer, pp. 19–28.

[12] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *ACM SIGARCH Computer Architecture News* (2008), vol. 36, ACM, pp. 26–35.

[13] BUELL, J., HECHT, D., HEO, J., SALADI, K., AND TAHERI, R. Methodology for performance analysis of vmware vsphere under tier-1 applications. *VMware Technical Journal 2*, 1 (2013), 19–28.

[14] CHANG, C.-R., WU, J.-J., AND LIU, P. An empirical study on memory sharing of virtual machines for server consolidation. In *ISPA* (2011), IEEE, pp. 244–249.

[15] CHIANG, J.-H., LI, H.-L., AND CHIUEH, T.-C. Introspection-based memory de-duplication and migration. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 51–62.

[16] FANG, Z., ZHANG, L., CARTER, J. B., HSIEH, W. C., AND MCKEE, S. A. Reevaluating online superpage promotion with hardware support. In *HPCA* (2001), IEEE.

[17] GANDHI, J., BASU, A., HILL, M. D., AND SWIFT, M. M. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture* (2014), IEEE Computer Society, pp. 178–189.

[18] GORMAN, M., AND HEALY, P. Supporting superpage allocation without additional hardware support. In *Proceedings of the 7th international symposium on Memory management* (2008), ACM, pp. 41–50.

[19] GORMAN, M., AND HEALY, P. Performance characteristics of explicit superpage support. In *International Symposium on Computer Architecture* (2010), Springer, pp. 293–310.

[20] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM 53*, 10 (2010), 85–93.

[21] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (2010), FAST'10.

[22] KWON, Y., YU, H., PETER, S., ROSSBACH, C. J., AND WITCHEL, E. Coordinated and efficient huge page management with ingens. In *OSDI 16* (2016), USENIX Association, pp. 705–721.

[23] MERRIFIELD, T., AND TAHERI, H. R. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2016), VEE '16, ACM.

[24] MILLER, K., FRANZ, F., RITTINGHAUS, M., HILLENBRAND, M., AND BELLOSA, F. Xlh: More effective memory deduplication scanners through cross-layer hints. In *USENIX Annual Technical Conference* (2013), pp. 279–290.

[25] NAVARRO, J., IYER, S., DRUSCHEL, P., AND COX, A. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 89–104.

[26] NAVARRO, J. E. *Transparent operating system support for superpages*. PhD thesis, Rice University, 2004.

[27] PHAM, B., VESELÝ, J., LOH, G. H., AND BHATTACHARJEE, A. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Micro architecture* (2015), ACM, pp. 1–12.

[28] SHARMA, P., AND KULKARNI, P. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing* (2012), ACM, pp. 15–26.

[29] SINDELAR, M., SITARAMAN, R. K., AND SHENOY, P. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures* (2011), ACM, pp. 367–378.

[30] WALDSPURGER, C. A. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.