Towards High Performance and Efficient Memory Deduplication via Mixed Pages

Lulu Yao, Yongkun Li, Fan Guo, Si Wu, Yinlong Xu, and John C.S. Lui, *Fellow, IEEE*

Abstract—Large pages are widely supported in modern hardware and OSes to reduce the overhead of TLB misses. However, memory deduplication can be inefficient with large pages, leading to low memory utilization. To simultaneously enjoy the benefits of high performance by accessing memory with large pages (e.g., 2 MB pages) and high deduplication rate by managing memory with base pages (e.g., 4 KB pages), we propose Smart Memory Dedupleiation (SmartMD), which is an adaptive and efficient memory management scheme via mixed pages. Specifically, we propose lightweight schemes to periodically monitor pages' access frequency and repetition rate, and present an adaptive conversion scheme to selectively split or reconstruct large pages. We further optimize SmartMD by developing SmartMD$^+$, which dynamically adjusts the page scanning cycle by monitoring the TLB miss cost, and reconstructs the split large pages in an on-demand way so as to reduce the CPU overhead of SmartMD. We further implement a prototype system and conduct extensive experiments with various workloads under different system settings. Experiment results show that SmartMD and SmartMD$^+$ can simultaneously achieve high access performance similar to systems using large pages, and achieve a deduplication rate similar to that applying aggressive deduplication scheme (i.e., KSM) on base pages.

Index Terms—Memory management, virtual memory, memory deduplication, large pages, virtualization

1 Introduction

In modern operating systems, processor accesses to memory typically use page tables to translate virtual addresses to physical addresses. To accelerate the translation, TLB was introduced to cache virtual-to-physical address mappings. TLB has a critical impact on system performance, its misses hinder performance by causing significant latency and additional memory accesses to page tables [2], [3], [4]. For example, TLB misses may increase memory accesses for page table walks by 20-40% [5], or even increase the execution time of some applications by 50% [6], [7].

Furthermore, in a hypervisor-based virtualization system, the hypervisor and guests maintain separate page tables, and TLB misses will lead to high-latency two-dimensional (virtual machine/guest dimension and host dimension) page table walks. Previous works [8], [9] show that this is often the primary contributor to the performance difference between virtualized and bare-metal systems, for example, two-dimensional address translation increases the TLB misses penalty up to 6$\times$ compared to native execution [10], [11]. In fact, the overhead of TLB misses has become one of the primary bottlenecks of memory access [4]. Moreover, while memory size becomes increasingly larger, TLB's capacity cannot grow at the same rate as DRAM [4], [12]. 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, X86 architectures provide the support of 4 KB, 2 MB, and 1 GB pages, and ARM architectures also support 4 KB, 1 MB, and 16 MB pages [13].

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., 2MB-pages), our experiments show that it is hard to find duplicate large pages even the memory contains a large amount of redundant data (see Table 2). In other words, memory deduplication in the unit of 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., 2 MB-pages) to base pages (e.g., 4KB-pages) and performs deduplication among base pages [15]. 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 last level TLB miss ratio and the amount of page table walks, thus degrading memory 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.

- Lulu Yao and Fan Guo are with the University of Science and Technology of China, Hefei 230026, China. E-mail: {luluyao, lps56}@mail.ustc.edu.cn.
- Yongkun Li, Si Wu, and Yinlong Xu are with the Anhui Province Key Laboratory of High Performance Computing, University of Science and Technology of China, Hefei 230026, China. E-mail: {ykli, siwu5938, ylxu}@ustc.edu.cn.
- John C.S. Lui is with the The Chinese University of Hong Kong, Hong Kong. E-mail: cslui@cse.cuhk.edu.hk.

Manuscript received 10 October 2021; revised 13 June 2022; accepted 3 July 2022. Date of publication 18 July 2022; date of current version 13 March 2023.
(Corresponding author: Yongkun Li.)
Recommended for acceptance by D. Gizopoulos.
Digital Object Identifier no. 10.1109/TC.2022.3191742

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, which aims for maximizing memory saving while keeping high performance of memory access. The main idea is to scan pages periodically and split cold large pages with high repetition rate to save memory space by memory deduplication, and at the same time, to reconstruct split large pages when they become hot to improve memory access performance. Due to the need of scanning memory periodically, SmartMD may incur extra CPU overhead. To address this issue, we further extend SmartMD by developing an optimized version SmartMD$^+$. SmartMD$^+$ supports monitoring the cost of TLB misses, which is defined as the percentage of page table walks, for each VM and the host system, and these information are used to dynamically adjust the page scanning cycle so as to balance the VMs' performance and memory saving. It also uses an on-demand policy to reconstruct large pages so as to reduce CPU overhead brought by SmartMD. 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. Specifically, 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 greatly accelerates the deduplication.
- 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 design two flexible adjustment strategies based on TLB miss ratio. Specifically, we propose a dynamic adjustment of the page scanning cycle according to the monitoring results of the TLB miss ratio of the whole system. Furthermore, we reconstruct large pages on-demand based on the monitored TLB miss ratio of each VM so as to guarantee the performance of VMs while reducing SmartMD's CPU overhead.
- We implement a prototype and conduct extensive experiments to show the efficiency of SmartMD and SmartMD$^+$. Results show that both of them 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. Meanwhile, SmartMD$^+$ further reduces the CPU overhead of SmartMD. We release the source code of SmartMD at https://github.com/ustcadsl/SmartMD.

2 Background and Motivation

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) [16] have been introduced. In this paper, we focus on Intel's extended page tables, while the design and conclusions are also applicable to systems using nested page tables. With the extended page tables, a VM will carry out a two-dimensional page walk to access its data with two steps. First, GVA are translated to their corresponding GPA using guest's page tables. Second, GPA are translated to their corresponding HPA using extended page tables.

When using base pages (i.e., 4 KB 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 [16], [17], which is apparently unacceptable. A common practice to accelerate the address translation is to cache frequently used global mapping from GVA to HPA in the TLB [18]. However, the page tables consistently grow as memory capacity increases, and this exacerbates the reduction of the TLB hit ratio, and finally degrades memory access performance.

2.2 Advantage of Using Large Pages

To increase the hit ratio of TLB and speedup the address translation in a system with a large amount of memory, large pages have been widely adopted in today's systems. Specifically, a large page is composed of a fixed number of contiguous base pages. For example, in a X86-64 system, OS uses one 2 MB-page entry to cover a contiguous 2 MB region of memory for its address translation, instead of using 512 4 KB-page entries to cover it. In a virtual environment, large pages can be supported in both guest's page tables and extended page tables [16]. With large pages, the page table becomes significantly smaller, and much larger memory space can be covered by a TLB of the same size. In this way, using large pages helps to increase TLB hit ratio, e.g., it reduces maximum number of memory references in a 2D page walk after a TLB miss from 24 to 15 [16].

TABLE 1
Performance Improvement With Large Pages

Benchmark	Host: Base Guest: Large	Host: Large Guest: Base	Host: Large Guest: Large
SPECjbb2005	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

The performance is normalized to the case of running the benchmark on the system using base pages in both guest and host OSes. Details of the benchmark programs are described in Section 4.

To show the improvement of memory access performance with large pages, we run experiments with various benchmarks (see Section 4 for detailed configurations of the experiments). 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 up to 68% over the baseline system in which only base page is used.

2.3 Memory Deduplication

It is common to have redundant data (i.e., same OSes or similar applications) residing in the memory of a virtualized machine [14]. For example, Difference Engine [19] reported that it can achieve up to 50% memory saving and VMware [20] also reported about 40% memory saving. Memory deduplication among different VMs is an efficient way to lower the memory demand and keep memory from being overcommitted, and many modern OSes enable this feature, e.g., Kernel Samepage Merging (KSM) in Linux [21] and Transparent Page Sharing (TPS) in VMware ESX server [20].

Memory deduplication schemes mainly adopt content-based page sharing (CBPS). Its key idea is to deduplicate multiple pages of the same content and keep only one physical copy, and make multiple virtual addresses point to this copy for sharing by modifying the page table entries. Taking KSM as an example, its core process is to periodically scan the memory area marked as deduplicable to find out and deduplicate redundant pages. As shown in Fig. 1, the workflow of KSM is as follows: (1) *Scan pages.* The KSM thread sequentially scans the memory area with the `MADV_MERGEABLE` tag starting from the first page. (2) *Search pages with the same content from the stable tree.* KSM maintains a red-black tree, called stable tree, for pages that have been deduplicated and are shared by multiple processes. For each scanned page, KSM first searches the same content page from the stable tree by comparing with each page in the tree in a byte-by-byte manner, and the scanned page is a duplicate page once the comparison matches. For pages with the same content, KSM will merge them. If a page with the same content is not found in the stable tree, KSM keeps searching in the unstable tree. (3) *Search pages with the same content from the unstable tree.* KSM also maintains a red-black tree named unstable tree, which keeps the candidate pages whose contents have not been changed in the previous two consecutive scans. Before searching the unstable tree, KSM hashes the whole page content to get a 32-bit checksum or signature. If the signature differs from the calculated hash in the last scan period, then the page content must have changed recently and it is skipped for deduplication. Otherwise KSM will look for the same page in the unstable tree, if it finds the identical page then perform the next step and inserts the scanned page into the unstable tree if not. (4) *Merge same pages.* After finding the same page from the stable tree or unstable tree, KSM first determines whether the scanned page is a large page. For each large page, it will be split into base pages first before subsequent processing [15], even in the recent 5.17-rc5 kernel release, which we call the aggressive deduplication approach (ADA). Next, KSM locks the scanned page and adds write protection to it, then the pages are compared byte by byte to confirm that their contents are identical. Finally, KSM calls the `replace_page` function to replace the scanned page with a page from the stable or unstable tree to realize page merge. In addition, if the merged page comes from an unstable tree, it is moved from the unstable tree to the stable tree.

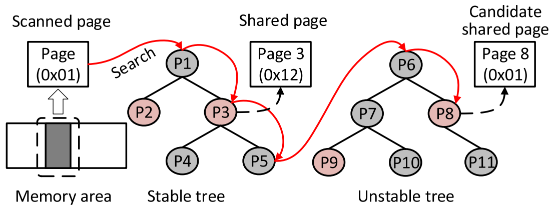


Fig. 1. Searching same content pages in KSM.

2.4 Impact of Large Pages on Memory Deduplication

Unfortunately, using large pages has a significant impact on memory deduplication, and in particular, there is a tradeoff between memory access performance and deduplication rate. Specifically, even though there are plenty of redundant data in the memory, there are few duplicate large pages. As a result, if deduplication is performed in unit of a large page, it may not be effective in removing redundant data and saving memory.

To further demonstrate the inefficiency of memory deduplication with large pages, we run experiments to show the amount of memory saving of two policies: deduplication in the unit of large pages and deduplication with only base pages via ADA. Our experiments show that ADA can save 13.7% - 47.9% of used memory for the benchmarks we studied, but deduplication in the unit of large page saves only 0.8% - 5.9% of used memory (see Table 2). That is, deduplication using large pages limits memory saving.

Besides the impact on memory saving, the above two deduplication policies also have a significant impact on memory access performance, or particularly, the TLB hit ratio. Specifically, by using ADA, large pages are all split into base pages, and this makes the page table become much larger, which finally reduces the hit ratio of TLB and increases the page table walks. To illustrate this, we also compare the percentage (abbreviated PCT) of page table walks with and without ADA of four VMs by running Graph500 and SPECjbb2005 (the detailed experiment

TABLE 2
Memory Saving and Performance of Large-Page-Based Systems With/Without ADA

Policy	Benchmark	Memory Saving	Performance
Dedup. with Huge Pages	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
Dedup. with Small Pages	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

Memory saving is normalized to the memory demand in the system without using any deduplication. The performance is normalized to that of the system using large pages without ADA.

configuration is described in Section 4). Note that the PCT of page table walks is defined as the percentage of CPU cycles for page table walks normalized to the total CPU cycles of running each application over a period of time. Note that we have similar conclusions for other benchmarks and we chose these two applications as examples. We record the PCT of page table walk of an application every five seconds and the results are shown in Fig. 2,we can see that the PCT of page table walks without ADA is always less than 1%, while the PCT of page table walks with ADA increases during the execution of deduplication, as more pages are split into base pages. In addition, ADA causes a dramatic increase in Graph500's page table walk percentage (i.e., up to 60%), thus having larger performance degradation. In the worst case, if ADA splits a large page that has high access frequency and low repetition rate, then it must compromise memory access performance, but brings limited memory saving.

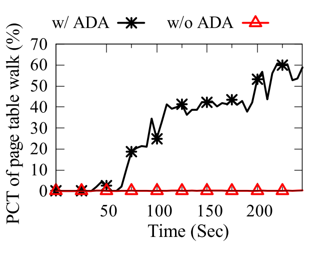
2.5 Motivation

As analyzed in the above subsection, there is a trade-off between memory access performance and memory deduplication rate when using large pages. Specifically, as shown in Table 2, with ADA (i.e., deduplication with base pages), we can save 13.7%-47.9% of memory space, but the memory system is slowed down by up to 30.5% due to increased TLB misses and page table walks 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.

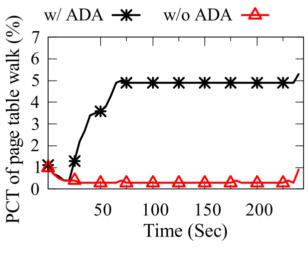
To address the above trade-off, we first carry out experiments to show the statistics of memory pages for all of our benchmarks. Our observation is that a large amount of large pages usually have high access frequency, but they have very few duplicate subpages, and vice versa.

Fig. 3 shows the distributions of large pages with high access frequency or high repetition rate of a VM which runs SPECjbb2005 for example in the interest of space. From Fig. 3a, we can see that the SPECjbb2005 benchmark constantly accesses some large pages throughout its entire run time while other large pages are rarely accessed. Fig. 3b shows that majority of large pages with high repetition rate appears only in few memory regions. Comparing Fig. 3a with Fig. 3b, 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. Note that ADA ignores the status of pages, but simply splits large pages without considering page access frequency and repetition rate. We emphasize that we have also conducted experiments with other benchmarks and the results show similar trends. We also test other repetition rates, and the results show that SmartMD can achieve good results for all benchmarks we studied by setting the repetition rate as 1/8.

To this end, our idea is to leverage the above observation and develop a memory system with mixed pages so as to simultaneously achieve high memory access performance and also high memory saving. To realize it, we propose a deduplication scheme which leverages the access frequency and repetition rate of pages. Specifically, we split only large

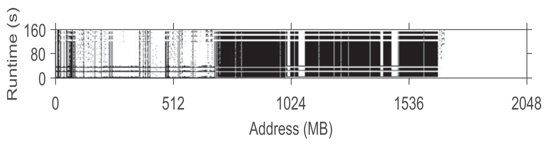


(a) Application: Graph500

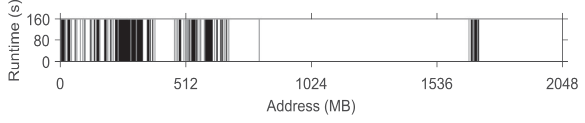


(b) Application: SPECjbb2005

Fig. 2. The percentage of page table walks with (w/) and without (w/o) ADA in every 5 seconds.



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



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

Fig. 3. Memory usage of SPECjbb2005.

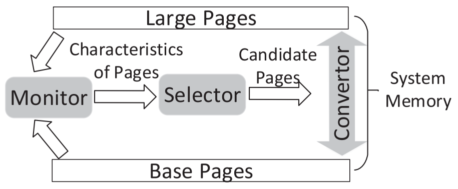


Fig. 4. Illustration of SmartMD's architecture.

pages with high repetition rate and low access frequency and perform deduplication among their subpages to save more memory while maintaining high access performance.

2.6 Challenges

Lightweight Monitoring of Pages' Status. To allow mixed pages, we need to track the access frequency and repetition rate of all pages, which are not directly disclosed by current OSes. Monitoring these parameters will introduce additional overheads, so we need to design efficient mechanisms with low overhead to track pages' status.

Adaptive Page Conversion. Incorrect page splitting will increase the TLB miss ratio and inappropriate page reconstruction will produce considerable useless page copies. Therefore, splitting large pages into base pages and reconstructing base pages into large pages may have significant negative impact on memory access performance. Thus, we 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 we need 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.

Efficient Reconstruction of Large Pages. Note that major OSes support splitting large pages into base pages to produce more opportunities for deduplication. However, they do not support the reconstruction of broken large pages which have been split and whose subpages are shared with subpages of other large pages [15], [22]. When a split large page turns to be hot and exhibits few redundant data, OSes cannot reconstruct it to obtain a better access performance. Furthermore, the locking operations during the reconstruction of large pages may substantially compromise system performance. Thus, we need to propose an approach to efficiently reconstruct large pages to improve memory access performance. However, the implementation of such an approach can be challenging, because it 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 difficulty on the reconstruction process.

3 Design of SmartMD and SmartMD$^+$

To achieve both high access performance and high deduplication rate, we design an efficient memory management system, SmartMD, which maintains mixed pages by dynamically converting between large pages (i.e., 2M-pages) and base pages (i.e., 4K-pages) according to the access frequencies and repetition rates of memory pages. In this section, we first present the design details of SmartMD, and then introduce its design optimizations, SmartMD$^+$.

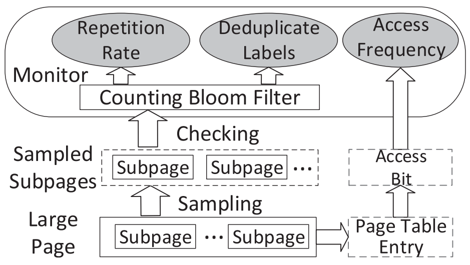


Fig. 5. Design of the monitor.

3.1 Overview of SmartMD

As shown in Fig. 4, SmartMD has three modules, *Monitor*, *Selector*, and *Converter*. In the Monitor, we periodically scan all large pages and base pages to record their access characteristics. Specifically, we provide two lightweight schemes to track the access frequency of pages and the number of duplicate subpages, or the repetition rate for large pages. This information will be used by the Selector to select candidate large pages for splitting or candidate base pages for reconstruction. In particular, we propose an algorithm which dynamically performs the selection according to the current system memory utilization, data access frequency, and large-page repetition rate. Finally, the Converter performs the conversion between large pages and base pages.

3.2 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. 5 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 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 the frequency of a page to $N/2$ when the system starts.

Detecting Repetition Rate of Pages. As mentioned before, SmartMD takes into account memory savings and memory access performance by splitting large pages with low access frequency and high repetition rate. Therefore, the percentage of duplicate base pages in a large page (denoted by *repetition rate*) is an important performance metric for potential

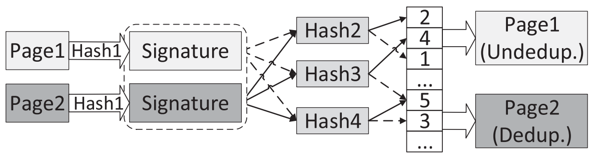


Fig. 6. Identification of duplicate pages by using a counting bloom filter.

memory savings. To measure the repetition rate of a large page, existing approaches (e.g., KSM) use comparison trees to identify duplicate pages [21]. However, these approaches are mainly used in deduplication and they carry high CPU overhead due to extensive page comparisons. For repetition rate measurement, we only need to determine whether the subpages in a large page are duplicates or not. Therefore, SmartMD uses a counting bloom filter for approximate identifications to reduce overhead.

The counting bloom filter is a one-dimensional vector, and each of its entries is a 3-bit counter. As shown in Fig. 6, 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 first generates a 32-bit signature, which is produced by applying a hash function (i.e., `Hash1` in Fig. 6) on its content and recorded to represent the page (see Section 2.3). Next, SmartMD applies three hash functions (i.e., `Hash2-Hash4`) on the subpage's 32-bit signature to calculate the indexes of its corresponding counters. Compared with the method of directly hashing the entire subpage (e.g., 4 KB), our scheme of hashing the signature of the subpage (i.e., 32-bit) is able to save the CPU overhead and mark the access characteristics of the subpages more instantly. After the above two steps, SmartMD can get the three corresponding indexes of a page in the counting bloom filter and increase the corresponding counters/entries by one. Since each entry in the bloom filter has 3 bits, the maximum count can be 8. If a page is checked for the first time (i.e., its recorded signature is not found), SmartMD will increase its corresponding counters in the bloom filter by one. Otherwise, if all of the counters are greater than one, we consider this page as a duplicate one. For example, `Page2` in Fig. 6 has all three counters in the counting bloom filter greater than one, so `Page2` is a duplicate page, while `Page1` has one counter whose value is one, so it is a unique page. 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.

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% [23], [24].

Despite using a secondary hashing strategy to speed up the calculation of the repetition rate of a large page, we still need to apply hashing to all subpages of a large page. This hinders further acceleration of the duplicate identification. Moreover, 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. Moreover, SmartMD does not allocate extra memory space to store signature and duplicate label of each base page, and store them in the metadata maintained by KSM.

SmartMD's sampling-based detecting algorithm can help to substantially reduce the CPU overhead for most workloads. Experiments show that the ratio of mis-identification 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 in 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 deduplication trees by only splitting large pages with high repetition rate.

3.3 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

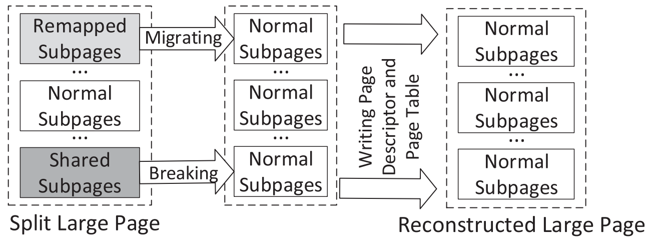


Fig. 7. Process of reconstructing split large pages.

more than 95% of duplicate subpages and split 40% fewer large pages than traditional aggressive deduplication approach.

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.

3.4 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 [15], [22]. We implement a utility in SmartMD to reconstruct split large pages. Fig. 7 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 a contiguous memory region 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 the 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.

3.5 SmartMD$^+$

SmartMD is able to achieve both considerable memory saving and high memory access performance, and it can be further optimized. In the following, we first analyze the limitations of SmartMD, and then present the optimizations of SmartMD$^+$.

Unawareness of TLB Miss Cost. SmartMD conducts conversion between large pages and base pages without considering the change of the TLB miss ratio. However, the conversion from large pages to base pages may increase the TLB miss ratio and the number of page table walks (see Section 2.4), which severely degrades the performance of VMs. To further enhance the performance of VMs, we can monitor the TLB miss cost (i.e., the percentage of page table walks) and use it as an input to intelligently adjust some settings of SmartMD.

Static Configuration of Scanning Cycle.SmartMD lacks elasticity in tuning the page scanning cycle. Note that SmartMD scans pages periodically after a fixed interval (i.e., $check\_interval$) to determine the degree of hotness of the pages (see Section 3.2). However, the page scanning interval is assumed to be set manually and statically. The static scanning interval may hurt the system performance in terms of the memory savings, the memory access performance or the CPU overhead introduced. On one hand, a large $check\_interval$ value results in lower CPU overhead and lower memory savings owing to that there is longer sleep time between successive scanning and there are less opportunities for the Monitor to discover short-lived and cold

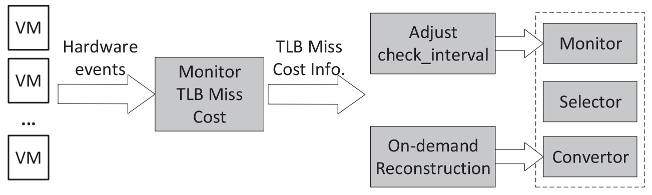


Fig. 8. Architecture of SmartMD$^+$.

large pages with high repetition rate. On the other hand, a small interval means that more large pages can be split into base pages, which incurs greater TLB contention and thus reduces the memory access performance. We refer the readers to the experiments in Section 4.2, which show the impact of the $check\_interval$ value on the access performance and memory saving.

Coarse-Grained Reconstruction. SmartMD reconstructs large pages without differentiating the performance of each VM. Note that SmartMD monitors the access frequency of split large pages and reconstructs large pages if they have high access frequency (see Section 3.4). However, this scheme ignores the differences among VMs (e.g., the different cache miss ratios, the differed memory access locality). Monitoring and reconstructing split large pages for a VM with low cache miss ratio is unnecessary and causes extra CPU overhead.

To further improve the elasticity and reduce the overhead of SmartMD, we propose an improved version, SmartMD$^+$. SmartMD$^+$ introduces several optimization techniques to address the above analyzed issues in SmartMD. First, SmartMD$^+$ supports monitoring the TLB miss cost, which is an important metric to measure large pages' impact on VMs' performance. Second, SmartMD$^+$ dynamically adjusts the scanning interval (i.e., $check\_interval$) to balance the memory access performance of VMs and the memory savings. Third, SmartMD$^+$ supports on-demand reconstruction of split large pages, which differentiates the performance variations of different VMs and then realizes fine-grained page reconstruction, so as to reduce the CPU overhead. Fig. 8 illustrates the techniques used in the SmartMD$^+$, which are elaborated as follows.

Monitoring TLB Miss Cost. To support monitoring the cost of TLB misses of each VM, we first acquire some hardware events from CPUs' Performance Monitoring Units (i.e, PMUs), including the total CPU cycles of each VM and the CPU cycles used for handling TLB misses (i.e., the CPU cycles spent for page table walking) in each VM. Then, we can obtain the percentage of CPU cycles which are used for TLB misses (denoted as TLB_{cost}) of each VM. Using a similar approach, we can monitor and calculate the TLB_{cost} of the whole system. We acquire hardware events once in a second to reduce the overhead of monitoring.

Dynamic Adjustment of Scanning Cycle. SmartMD manually sets the value of the scanning interval (i.e., $check\_interval$), which cannot be adjusted and thus is inefficient to meet the dynamic system demands and states. To address this issue, SmartMD$^+$ supports dynamic adjustment of the scanning interval. Specifically, SmartMD$^+$ sets two thresholds about the costs of the TLB misses, i.e., TLB_{low} and TLB_{high}. If TLB_{cost} of whole system is lower than TLB_{low}, indicating that many large pages can be further split into base pages, then we decrease $check\_interval$ by 500 ms to allow more cold large pages (with high repetition rate) to be split into base pages to save more memory. If TLB_{cost} of whole system is larger than TLB_{high}, then we increase $check\_interval$ by 500 ms to obtain more hot large pages, such that the overall memory access performance can be improved.

On-Demand Reconstruction of Large Pages. SmartMD$^+$ adopts a fine-grained reconstruction mechanism based on the TLB miss cost (i.e., TLB_{cost}) of each VM. It sets a threshold TLB_{thres} about the TLB miss cost, and only monitors and reconstructs split large pages in the VMs whose TLB_{cost} is larger than TLB_{thres}. For the VMs with TLB_{cost} smaller than TLB_{thres}, we do not need to monitor the split large pages and the reconstruction operations are also eliminated. This mechanism improves the performance for VMs that have high TLB miss costs, and at the same time, save the CPU overhead for VMs with low TLB miss costs.

4 Evaluation

To show the efficacy and efficiency, we implement a prototype of SmartMD and its improved version SmartMD$^+$ on Linux 3.4 and conduct experiments using QEMU to manage KVM. As described in Section 3.5, the major difference between SmartMD$^+$ and SmartMD is that SmartMD$^+$ is more intelligent by adjusting the $check\_interval$ value automatically and performing page conversion on demand, while other functions are essentially the same. Therefore, our experiments mainly focus on evaluating SmartMD (with fixed $check\_interval$ value), while we also conduct some evaluations on SmartMD$^+$ to show its effectiveness. Our experiments run on a NUMA (non-uniform memory access) server consisting of two physical NUMA nodes with two physical CPUs. Each NUMA node has an Intel Xeon E5-2650 v4 2.20 GHz CPU with twelve CPU cores, 32 GB DRAM. Our server mounts a 2 TB hard disk (WD20EFRX), and 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 4 GB RAM, and all VMs are hosted on different CPU cores in the same NUMA node. In our experiments, we focus only on 2 MB and 4 KB pages, which are commonly used in most applications. We run the following benchmark programs in each VM.Note that their memory demands without deduplication also include the guest OS.

- *Graph500 [25].* 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 per second as the performance metric of the benchmark. Memory usage: 2.7 GB.
- *SPECjbb2005 [26].* SPECjbb2005 is a benchmark for evaluating performance of server-side Java business applications. We run SPECjbb2005 in each VM and use the average bops (business operations per second) of all VMs as its performance metric. Memory usage: 1.7 GB.
- *Liblinear [27].* Liblinear is a suite of linear classifiers for a data set with millions of instances and features.

TABLE 3
Default Parameter Setting

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

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. Memory usage: 4.0 GB.

- *Sysbench [28].* 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. Memory usage: 2.93 GB.
- *Biobench [29].* Biobench is a suite of bioinformatics applications. We run Mummer, a program in Biobench on the human-chromosomes dataset [30], and measure its average execution time in VMs. Memory usage: 3.42 GB.

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 [15], which is one of the state-of-the-art scheme using mixed pages to make a trade-off between access performance and memory saving. The fourth one is HawkEye [31], which optimizes the management of huge pages, but still adopts the origin KSM for memory deduplication. Default values of the parameters used in the experiments are listed in Table 3. We adopt the same rate at which for the schemes to scan and identify duplicate pages for a fair comparison.

Note that with the adaptive page conversion scheme described in Section 3.4, 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 4.1, 4.2, 4.3, 4.4, and 4.5), we use fixed non-zero $Thresh_{cold}$ 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 SmartMD, we run experiments in a memory-constrained system (see Section 4.6). In particular, we limit the available memory space of the host by running hugetlbfs [32]. Pages held by hugetlbfs cannot be deduplicated or swapped out, so we can flexibly adjust size of the host's memory available for running benchmarks.

The improvements of SmartMD$^+$ over SmartMD are mainly divided into two aspects. One is that SmartMD$^+$ reduces the CPU overhead via the fine-grained reconstruction strategy, and the other is that SmartMD$^+$ simultaneously achieves the deduplication effect close to ADA and the memory access performance close to no-splitting method by dynamically tuning the page scanning cycle. Thereby, to evaluate the effectiveness of SmartMD$^+$, we compare it with SmartMD with manual settings. (see Section 4.7).

4.1 Overhead of SmartMD

CPU Overhead. Since Graph500 has the highest repetition rate of these applications and our goal is to study the CPU overhead of SmartMD in the high repetition rate scenario. We focus on Graph500 in this experiment and show the CPU overhead of SmartMD with the other two memory deduplication schemes. The results are shown in Table 4. Both the monitoring thread and deduplication thread use additional CPU cycles. No-splitting always scans and compares pages at the granularity of large pages, and the content of large pages is more likely to be changed than base pages, so fewer page comparisons are required and thus less CPU is used. KSM and HawkEye uses aggressive deduplication without tracking the 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 4, KSM spends more CPU time than No-splitting, Ingens and SmartMD by 45%, 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,

TABLE 4
Average CPU Utilization Sampled in Every Second

	Monitor thread	Dedup thread	Total
No-splitting	0	23.1%	23.1%
KSM/HawkEye	0	33.5%	33.5%
Ingens	5.3%	21.3%	26.6%
SmartMD	13.1%	11.9%	25.0%

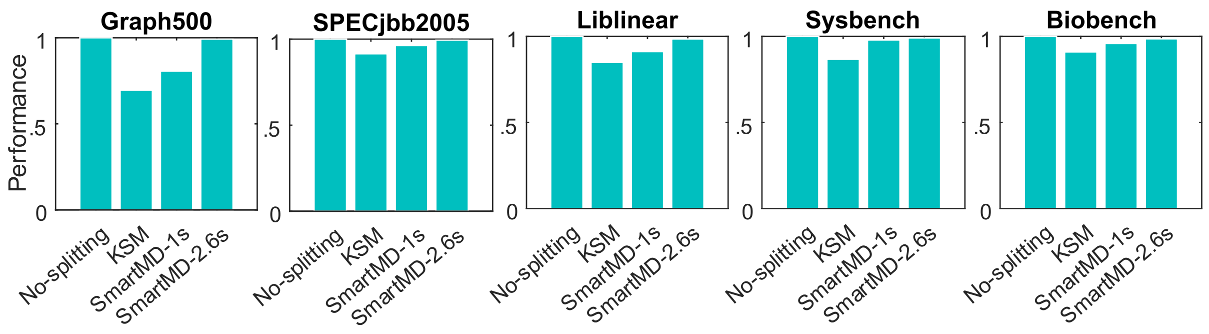


Fig. 9. Performance of the benchmarks under various deduplication policies.

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 result, 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 4.3. Note that for the scenarios where the repetition rate is not high, SmartMD still has benefits, for example, SmartMD uses bloom filter to filter out many useless page comparisons.

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 4 KB, the ratio of extra memory space used to store the filters is only $(3bits \times 8) \div (4\ KB) = 3/2^{12}$. For each large page, we use 32B to store its access frequency, repetition rate and some necessary pointers. Since the size of a large page is 2 MB, SmartMD requires additional $32B \div 2\ MB = 1/2^{16}$ of the memory space for large pages. Since SmartMD is implemented based on KSM, it can make full use of some metadata of KSM, such as the stable and unstable trees, the information about each scanned page (e.g., the checksum and virtual address). SmartMD itself requires only a small amount of metadata, e.g., the hotness of each page, the information about whether a page is a large page, etc. For example, 16 GB memory is used during the running of Liblinear on four VMs, while SmartMD needs only 12 MB to store bloom filers for base pages and 0.25 MB to store the metadata for large pages. Thus, the memory overhead of SmartMD is negligible.

4.2 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. By aggressively splitting any large pages to maximize deduplication opportunities, KSM can achieve the highest memory saving. 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.

We first show performance of the benchmarks by using SmartMD, KSM and no-splitting in Fig. 9, where we normalize the performance, whose metrics are introduced in the description of the benchmarks in Section 4, against that of the no-splitting. In the experiments, we use two different check _ interval values (1.0 s and 2.6 s) in SmartMD to vary the time period between resetting access bits and its next reaching of the bits. Accordingly, SmartMD is named SmartMD-1 s and SmartMD-2.6 s, respectively. Fig. 9 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 splitting necessary large pages.

The results of memory saving are shown in Fig. 10. 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 4× to 31× as large as the saving received in no-splitting. From Fig. 10, we can also see that 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 3.2 GB SmartMD-2.6 s and KSM take 118 s and 161 s, respectively.

Looking into Figs. 9 and 10, we can see that SmartMD takes both benefits on memory saving and access performance. Specifically, SmartMD can save 4× to 21× 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 9× 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 1.0 s to 2.6 s. 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

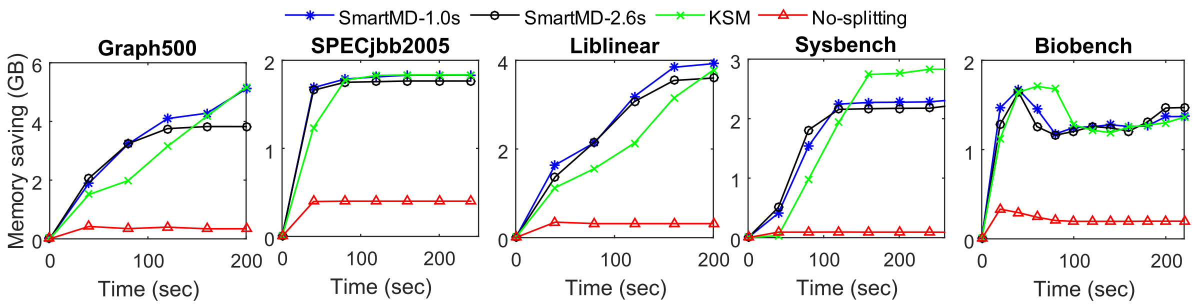


Fig. 10. Memory saving under various deduplication policies.

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.

In addition, we adjust the scale (23) and edgefactor (17) of Graph500 to consume more memory in order to analyze the effect of VMs/applications with different sizes. The experimental results show that SmartMD can still take both benefits of memory saving and access performance. We find that the performance of SmartMD-1 s, SmartMD-1.5s and SmartMD-2.6 s are very similar, and the difference in memory saving is not significant. This means that when increasing the memory footprint of VMs/applications, the $check\_interval$ needs to be adjusted to better balance the trade-off between access performance and memory saving.

4.3 Comparison With Ingens and HawkEye

Ingens [15] and HawkEye [31] aim to provide better support for large pages in current OSes. Specifically, Ingens enables conversion from base pages to large pages to maintain high memory access performance. It also selectively splits large pages for larger deduplication rate. However, in the selection of large pages, it considers only the access frequency and does not take into account the repetition rate. Besides, it does not consider the change of page access frequency to reconstruct large pages. HawkEye also optimizes huge page management using techniques like asynchronous page pre-zeroing and deduplication of zero-filled pages. However, it still adopts the aggressive deduplication approach. As Ingens and HawkEye are implemented based on Linux 4.3, for fair comparison, we also ported SmartMD to the same kernel version (Linux 4.3). The default values of the system parameters used in the experiments are listed in Table 3, note that some of the parameters are not used by Ingens and HawkEye. Table 5 shows the performance of the benchmark programs, and Fig. 11 shows the memory savings. First, compared with Ingens, SmartMD saves 1.3× to 3.5×

TABLE 5
Performance of SmartMD, Ingens and HawkEye

	Ingens	SmartMD	HawkEye
Graph500	0.988	0.995	0.733
SPECjbb2005	0.990	0.991	0.910
Liblinear	0.987	0.993	0.851
Sysbench	0.983	0.990	0.896
Biobench	0.977	0.985	0.921

Results are normalized to that of No-splitting.

memory, while keeping the same performance. Note that Ingens splits all large pages that are considered to be cold, so it has to throttle the generation of cold pages to keep the memory access performance close to the case of no-splitting. This is achieved by postponing the check of access bits, but it leaves fewer pages available for deduplication. SmartMD can more accurately identify the right large pages (with low access frequency and high repetition rate) for splitting, so it reduces the chance of doing unnecessary splitting. SmartMD also performs necessary reconstruction of large pages to keep high memory performance. Second, for HawkEye, we see that it still shows the trade-off between performance and memory saving, due to its use of aggressive deduplication, while SmartMD realizes the balance.

4.4 Impact of Linux Kernel

Multiple optimizations on memory deduplication are made in recent Linux kernel. For example, for the recent Linux 5.10, it has the following optimizations compared to Linux 3.4: (1) Optimization on the stable tree. In kernel 3.4, each node in the stable tree represents a shared page and contains the information required for reverse mapping from a shared page to virtual addresses that map this page. Linux 5.10 maintains two types of nodes in the stable tree to avoid high latency of the reverse mapping walks on shared pages. (2) Transparent huge page (THP) management. THP also works for tmpfs and shared memory (i.e., shmem), instead of only the anonymous memory mappings, and it is also supported in NUMA architectures. Besides, Linux 5.10 also supports more defragmentation policies, such as `defer` and `defer+madvise`.

We also port SmartMD and HawkEye to Linux 5.10, and we point out that Ingens cannot be easily ported to Linux 5.10, because the Linux kernel makes a lot of changes for various modules in version 5.10, such as the radix tree, memory cgroup, memory deduplication, transparent huge page management, memory defragmentation module, etc., while the implementation of Ingens is closely coupled with these modules. The results evaluated on Linux kernel 5.10 are shown in Fig. 12. First, SmartMD still achieves a better trade-off between memory saving and access performance. Second, the performance gap between No-splitting and KSM becomes smaller (10%) compared to the results shown in Fig. 10, and this implies that the optimizations of huge page management and deduplication in recent kernel also improves the memory access performance. However, by adjusting the parameters of KSM and THP (e.g., `max_page_sharing` and `stable_node_chains_prune_millisecs` in KSM and `defrag` in

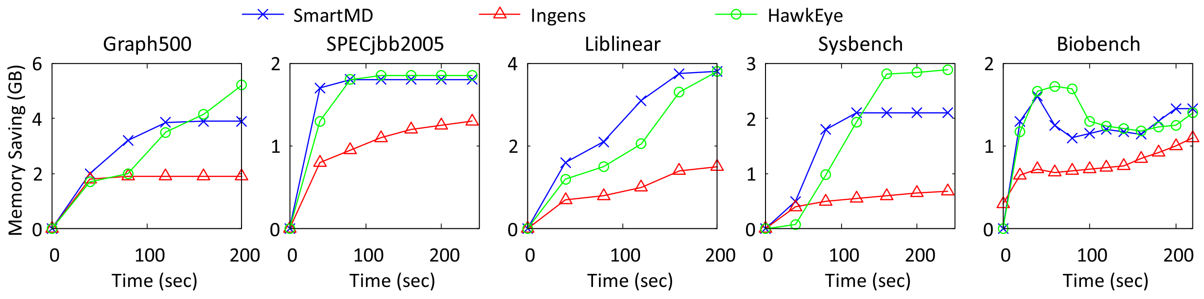


Fig. 11. Comparing the memory saving of SmartMD with Ingens and HawkEye.

THP), we find that the gap between No-splitting and KSM can be up to 20% or even larger. Besides, by studying more benchmarks, we also find that the gap between No-splitting and KSM may still be large for some benchmarks like SPECjbb2005, e.g., the gap is almost the same with the result evaluated on old kernels. Thus, the optimizations in recent kernels do not fully address the performance issues caused by deduplication, and SmartMD is still necessary to balance the trade-off. Finally, we find that HawkEye has a consistent performance with KSM even for Linux 5.10 because it uses the same aggressive deduplication strategy like KSM.

4.5 Performance in NUMA Environment

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

To study the performance impact of the NUMA architecture, we place two VMs on one NUMA node, and another two on a different node and re-run the benchmarks with SmartMD. The performance results are shown in Table 6. 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 deduplicates 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.

4.6 Performance in Memory Over-Committed Systems

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. In the interest of space, we take Graph500 and Liblinear as examples to show, and the conclusions also hold for other three benchmarks. The results are shown in Fig. 13, 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 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 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 trade-off between memory saving and access performance under different levels of memory overcommitments.

4.7 Performance of SmartMD$^+$

In this section, we evaluate the memory saving and performance of SmartMD$^+$. We compare SmartMD$^+$ with SmartMD which has fixed *check_interval* value. Here, we also use two different *check_interval* values for SmartMD (i. e, 1.0 s and 2.6 s) to represent the near-optimal memory saving and the near-optimal benchmark performance when using SmartMD. The experimental results are showed in Figs. 14 and 15.

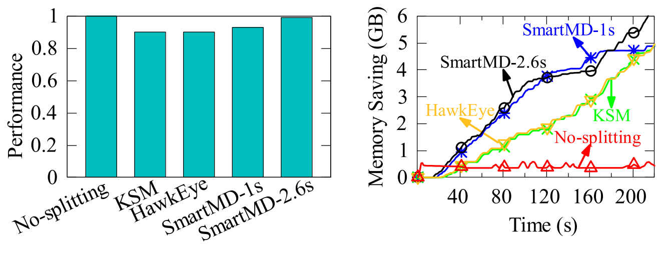


Fig. 12. Memory saving and performance of various deduplication policies when running Graph500 on Linux 5.10.

TABLE 6
Performance Impact of NUMA Architecture

	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%

The degradation ratio is calculated by comparing with No-splitting.

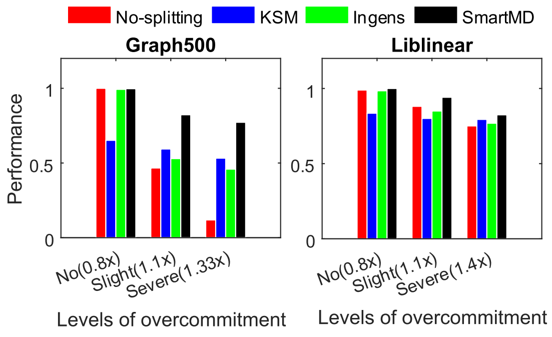


Fig. 13. Performance in overcommitted systems.

From Fig. 14, SmartMD$^+$ has the similar memory saving to SmartMD-1.0 s which shows the optimal memory saving effect, while it improves the memory saving of SmartMD-2.6 s by up to 29.4%. From Fig. 15, SmartMD$^+$ gains the similar benchmark performance to SmartMD-2.6 s that exhibits the optimal memory access performance, and it improves the benchmark performance of SmartMD-1.0 s by up to 42.3%. Note that we also conduct experiments with other three benchmarks and the results show the same conclusion. But the benefits of using SmartMD$^+$ are not obvious for these applications, mainly because both SmartMD-1.0s and SmartMD-2.6 s can achieve near-optimal results in these applications. Overall, SmartMD$^+$ can achieve near-optimal memory saving and near-optimal benchmark performance, simultaneously. The reason is that SmartMD$^+$ is capable of adaptively choosing the most appropriate $check\_interval$ values for different applications based on their TLB miss costs. It not only guarantees the performance of the benchmark programs, but also brings a lot of memory saving.

We further evaluate the CPU overhead of SmartMD$^+$ with Graph500, and compare it with SmartMD-2.6 s (see Section 4.1). The results are showed in Table 7. From the results, we can conclude that SmartMD$^+$ uses 3.7% fewer CPU cycles in its monitor thread. This is because that SmartMD$^+$ adopts an on-demand reconstruction mechanism, and only monitors and reconstructs the split large pages of the VMs which have high TLB miss costs. As a result, SmartMD$^+$ can efficiently save the CPU cycles for monitoring and reconstruction. A side effect is that SmartMD$^+$ uses slightly more CPU cycles in its deduplication thread, and this is because that there are more pages that need to be deduplicated in a shorter period of time.

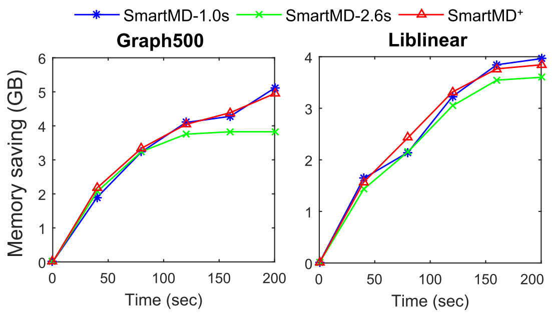


Fig. 14. Memory saving between SmartMD and SmartMD$^+$.

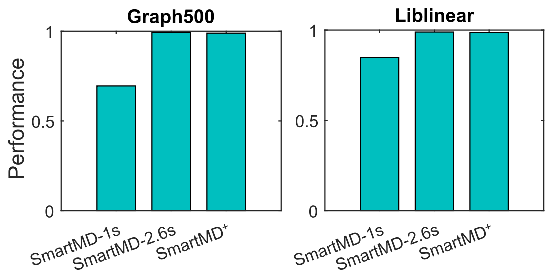


Fig. 15. Performance of SmartMD and SmartMD$^+$.

5 Related Work

Management of Large Pages. To efficiently use large pages, many researchers proposed schemes to manage pages of different sizes [11], [33], [34]. For example, Navarro *et al.* [35] provide a tool for FreeBSD to support multiple page sizes with contiguity-awareness and fragmentation reduction. Gorman *et al.* [36] 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 [37] proposes an API for applications to explicitly request huge pages. There are also some efforts to design hugepage-friendly memory allocation strategies [11], [33], [34], which aim to maximize hugepage coverage and minimize fragmentation overheads. Different from SmartMD, the above works do not consider memory deduplication.

Memory Deduplication. Memory deduplication has attracted a lot of attention [19], [20], [21], [38], [39]. Besides KSM [21] and TPS [20], there are many optimizations regarding memory deduplication. Most works focus on optimizing the deduplication algorithm to achieve faster deduplication speed, lower deduplication overhead, and higher memory savings, e.g., UKSM [40], AMT-KSM [41], CMD [42], etc. Besides, nukSM [43] optimizes the performance and fairness impact of memory deduplication in NUMA. The above works can be considered as aggressive deduplication schemes whose objective is to reduce memory usage and achieve higher deduplication rate. However, they do not consider the impact of large pages.

Some recent works also investigated deduplication in large page systems. Ingens [15] is a recently proposed memory deduplication scheme, and it provides a coordinated transparent huge page support for the OS and hypervisor. HawkEye [31] and MEGA [7] also support optimized deduplication with huge pages. Specifically, HawkEye [31] leverages a set of simple-yet-effective algorithms to address the performance, page fault latency and memory bloat issues of huge page management, and MEGA also tackles problems associated with huge pages, including increased page fault latency, memory bloating as well as memory fragmentation by using basic tracking mechanisms and a novel memory

TABLE 7
Average CPU Utilization Sampled in Every Second

	Monitor thread	Dedup thread	Total
SmartMD-2.6 s	13.1%	11.9%	25.0%
SmartMD$^+$	9.4%	12.4%	21.8%

compaction algorithm. However, both HawkEye and MEGA do not consider the tradeoffs of memory deduplication under mixed pages.

Different from the above works, SmartMD focuses on the trade-off of memory saving and performance when enabling deduplication in large page based systems. In particular, SmartMD differs from the above works in the following aspects. First, SmartMD selectively splits large pages according to their access frequency and repetition rate, while Ingens only considers pages' access frequency, HawkEye and MEGA aggressively split all large pages. second, 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. Finally, SmartMD adaptively selects pages for splitting and reconstruction, and uses sampling-based counting bloom filters and duplication labels to reduce CPU overhead of deduplication.

We also like to point out that the effectiveness of SmartMD is not affected by the X86 consistency model [44], which dictates TLB prefetches to set the access bits in the page table for the prefetched page table entries. Specifically, large pages that are split by SmartMD are accessed less frequently, so inaccurate prefetch has little impact on application performance. For base pages with high access frequency, SmartMD will merge them into large pages, which can greatly reduce TLB entries, and this brings benefit larger than the overhead caused by inaccurate prefetch. In addition, as the implementation of SmartMD is based on Linux KSM, and it simply reuses the KSM interface, SmartMD does not introduce new security vulnerabilities.

6 Conclusion

In this work, we proposed 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. We also designed SmartMD$^+$ via on-demand page reconstruction and tunable page scanning, and it further improves the performance and reduces the CPU overhead.

Acknowledgments

An earlier conference version of the paper appeared in Annual Technical Conference (ATC 2017) with a DOI of 10.5555/3154690.3154759. In this journal version, we extend SmartMD by addressing several issues (e.g., unawareness of TLB misses, inflexible configuration of scanning cycle and coarse-grained page reconstruction), and then propose an optimized version, SmartMD$^+$. We also add experiments to validate the effectiveness of SmartMD$^+$

References

[1] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. S. Lui, "SmartMD: A high performance deduplication engine with mixed pages," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2017, pp. 733–744.

[2] G. Vavouliotis *et al.*, "Exploiting page table locality for agile TLB prefetching," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Architecture*, 2021, pp. 85–98.

[3] N. Amit, A. Tai, and M. Wei, "Don't shoot down TLB shootdowns!," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 85–98.

[4] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2020, pp. 1093–1108.

[5] A. Bhattacharjee, "Translation-triggered prefetching," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2017, pp. 63–76.

[6] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proc. Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2013, pp. 237–248.

[7] T. Michailidis, A. Delis, and M. Roussopoulos, "MEGA: Overcoming traditional problems with os huge page management," in *Proc. 12th ACM Int. Conf. Syst. Storage*, 2019, pp. 121–131.

[8] J. Buell, D. Hecht, J. Heo, K. Saladi, and R. Taheri, "Methodology for performance analysis of VMware vSphere under tier-1 applications," *VMware Tech. J.*, vol. 2, no. 1, pp. 19–28, 2013.

[9] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 178–189.

[10] C. Alverti *et al.*, "Enhancing and exploiting contiguity for fast memory virtualization," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Architecture*, 2020, pp. 515–528.

[11] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2018, pp. 679–692.

[12] A. Bhattacharjee, "Preserving virtual memory by mitigating the address translation wall," *IEEE MICRO*, vol. 37, no. 5, pp. 6–10, Sep./Oct. 2017.

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

[14] C.-R. Chang, J.-J. Wu, and P. Liu, "An empirical study on memory sharing of virtual machines for server consolidation," in *Proc. IEEE 9th Int. Symp. Parallel Distrib. Process. Appl.*, 2011, pp. 244–249.

[15] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *Proc. 12th USENIX Conf. Oper. Syst. Des. Implementation*, 2016, pp. 705–721.

[16] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," *SIGARCH Comput. Architecture News*, vol. 36, no. 1, pp. 26–35, 2008.

[17] T. Merrifield and H. R. Taheri, "Performance implications of extended page tables on virtualized x86 processors," in *Proc. 12th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2016, pp. 25–35.

[18] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2015, pp. 1–12.

[19] D. Gupta *et al.*, "Difference engine: Harnessing memory redundancy in virtual machines," *Commun. ACM*, vol. 53, no. 10, pp. 85–93, 2010.

[20] C. A. Waldspurger, "Memory resource management in vmware ESX server," *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.

[21] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proc. Linux Symp.*, 2009, pp. 19–28.

[22] "The big khugepaged redesign," 2015. [Online]. Available: https://lwn.net/Articles/634384

[23] L. Fan, P. Cao, J. Almeida, and A. Z., "Summary cache: A scalable wide-area web cache sharing protocol," 1998. [Online]. Available: http://pages.cs.wisc.edu/cao/papers/summary-cache

[24] "Bloomfilter," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Bloom_filter

[25] "Graph500," 2010. [Online]. Available: https://graph500.org/?page id=12

[26] "SPECjbb2005," 2005. [Online]. Available: https://www.spec.org/jbb2005/

[27] "Liblinear," 2008. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/liblinear/

[28] "Sysbench," 2004. [Online]. Available: https://github.com/akopytov/sysbench

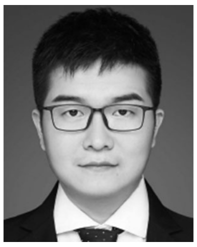
[29] K. Albayraktaroglu *et al.*, "BioBench: A benchmark suite of bioinformatics applications," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2005, pp. 2–9.

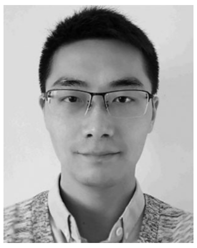
[30] "Human chromosomes," 1999. [Online]. Available: http://mummer.sourceforge.net/applications.html

[31] A. Panwar, S. Bansal, and K. Gopinath, "HawkEye: Efficient fine-grained os support for huge pages," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2019, pp. 347–360.
[32] "Hugetlbfs," 2022. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt
[33] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated page: Supporting fragmented memory allocation for large pages," in *Proc. 47th ACM/IEEE Annu. Int. Symp. Comput. Architecture*, 2020, pp. 913–925.
[34] A. Hunter, C. Kennelly, P. Turner, D. Gove, T. Moseley, and P. Ranganathan, "Beyond malloc efficiency to fleet efficiency: A hugepage-aware memory allocator," in *Proc. USENIX Conf. Oper. Syst. Des. Implementation*, 2021, pp. 257–273.
[35] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 89–104, 2002.
[36] M. Gorman and P. Healy, "Supporting superpage allocation without additional hardware support," in *Proc. 7th Int. Symp. Memory Manage.*, 2008, pp. 41–50.
[37] M. Gorman and P. Healy, "Performance characteristics of explicit superpage support," in *Proc. Int. Symp. Comput. Archit.*, 2010, pp. 293–310.
[38] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "XLH: More effective memory deduplication scanners through cross-layer hints," in *Proc. Conf. USENIX Annu. Tech. Conf.*, 2013, pp. 279–290.
[39] P. Sharma and P. Kulkarni, "Singleton: System-wide page deduplication in virtual environments," in *Proc. 21st Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2012, pp. 15–26.
[40] N. Xia, C. Tian, Y. Luo, H. Liu, and X. Wang, "UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 325–340.
[41] L. You, Y. Li, F. Guo, Y. Xu, J. Chen, and L. Yuan, "Leveraging array mapped tries in KSM for lightweight memory deduplication," in *Proc. IEEE Int. Conf. Netw. Architecture Storage*, 2019, pp. 1–8.
[42] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao, "CMD: Classification-based memory deduplication through page access characteristics," in *Proc. 10th ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environ.*, 2014, pp. 65–76.
[43] A. Panda, A. Panwar, and A. Basu, "nuKSM: NUMA-aware memory de-duplication on multi-socket servers," in *Proc. 30th Int. Conf. Parallel Architectures Compilation Techn.*, 2021, pp. 258–273.
[44] G. Vavouliotis, L. Alvarez, B. Grot, D. Jiménez, and M. Casas, "Morrigan: A composite instruction TLB prefetcher," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2021, pp. 1138–1153.

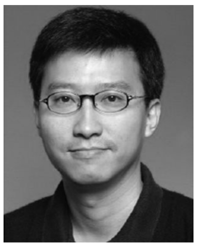
Lulu Yao received the bachelor's degree in computer science and technology from Zhengzhou University, in 2018. He is now working toward the PhD degree with the School of Computer Science and Technology, University of Science and Technology of China. His research interests include virtualization and operating system, especially in memory systems.

Yongkun Li received the BEng degree in computer science from the University of Science and Technology of China, in 2008, and the PhD degree in computer science and engineering from The Chinese University of Hong Kong, in 2012. He is currently an associate professor with the School of Computer Science and Technology, University of Science and Technology of China. His research mainly focuses on memory and file systems, including key-value systems, distributed file systems, as well as memory and I/O optimization for virtualized systems.


Fan Guo received the PhD degree from the University of Science and Technology of China, in 2019, and now works in the VirtAI Technology Corporation. His main research interests include memory and GPU virtualization, operating system, as well as memory support for machine learning.


Si Wu received the BEng and PhD degrees in computer science from the University of Science and Technology of China, in 2011 and 2016, respectively. He is now an associate researcher with the School of Computer Science and Technology, University of Science and Technology of China. His research interests include storage reliability and distributed storage.

Yinlong Xu received the BS degree in mathematics from Peking University, in 1983, and the MS and PhD degrees in computer science from the University of Science and Technology of China (USTC), in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology, USTC, and is leading a research group in doing some storage and high performance computing research. His research interests include network coding, storage systems, etc. He received the Excellent PhD Advisor Award of Chinese Academy of Sciences, in 2006.


John C.S. Lui (Fellow, IEEE) received the PhD degree in computer science from the University of California at Los Angeles. He is currently the Choh Ming Li chair professor with the CSE Department, The Chinese University of Hong Kong. His current research interests include machine learning, online learning (e.g., multi-armed bandit and reinforcement learning), network science, future Internet architectures and protocols, network economics, network/system security, and large-scale storage systems. He is an elected member of the IFIP WG 7.3, and a senior research fellow of the Croucher Foundation. He is a fellow of ACM and Hong Kong Academy of Engineering Sciences. He received various departmental teaching awards and the CUHK Vice-Chancellor Exemplary Teaching Award.