CMD: Classification-based Memory Deduplication through Page Access Characteristics

Licheng Chen[†§], Zhipeng Wei[†§], Zehan Cui[†§], Mingyu Chen[†], Haiyang Pan[†§], Yungang Bao[†]

[†]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[§]University of Chinese Academy of Sciences
{chenlicheng, weizhipeng, cuizehan, cmy, panhaiyang, baoyg}@ict.ac.cn

Abstract

Limited main memory size is considered as one of the major bottlenecks in virtualization environments. Content-Based Page Sharing (CBPS) is an efficient memory deduplication technique to reduce server memory requirements, in which pages with same content are detected and shared into a single copy. As the widely used implementation of CBPS, Kernel Samepage Merging (KSM) maintains the whole memory pages into two global comparison trees (a stable tree and an unstable tree). To detect page sharing opportunities, each candidate page needs to be compared with pages already in these two large global trees. However since the vast majority of pages have different content with it, it will result in massive futile page comparisons and thus heavy overhead.

In this paper, we propose a lightweight page Classification-based Memory Deduplication approach named **CMD** to reduce futile page comparison overhead meanwhile to detect page sharing opportunities efficiently. The main innovation of CMD is that pages are grouped into different classifications based on page access characteristics. Pages with similar access characteristics are suggested to have higher possibility with same content, thus they are grouped into the same classification. In CMD, the large global comparison trees are divided into multiple small trees with dedicated local ones in each page classification. Page comparisons are performed just in the same classification, and pages from different classifications are never compared (since they probably result in futile comparisons). The experimental results show that CMD can efficiently reduce page comparisons (by about 68.5%) meanwhile detect nearly the same (by more than 98%) or even more page sharing opportunities.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Main Memory; D.4.8 [*Performance*]: Measurements

Keywords Memory Deduplication; Page Classification; Page Access Characteristics

1. Introduction

Cloud computing becomes increasingly popular and competitive both in industry and academia. In cloud computing, multiple virtual machines (VMs) can be collocated on a single physical server, and they can operate independently with virtualization technology [16, 25], which is promising to provide flexible allocation, migration of services, and better security isolation. In the virtualization environment, physical resources (such as processor, main memory) are managed by a software layer called hypervisor (or Virtual Machine Monitor, VMM), and the primary goal of a hypervisor is to provide efficient resource sharing among multiple co-running virtual machines.

However, as the number of VMs collocated on a physical server keeps increasing (e.g. it can collocate up to 8 virtual machines on a physical core in desktop cloud environment), meanwhile with the increasing size of working set of applications running in a virtual machine, virtualization has placed heavy pressure on memory system for larger capacity. However, since the increasing speed of main memory capacity falls behind with the demand, the limited main memory size has been considered as one of the major bottlenecks to consolidate more number of guest VMs on a hosting server [17, 22]. Memory deduplication is an efficient technique to alleviate the memory capacity bottleneck, which detects and reduces page duplication to save memory. A large volume of prior work has shown that there are great opportunities in memory deduplicaion, e.g. Difference Engine [17] reported absolute memory savings of 50% across VMs, and VMware [29] reported about 40% memory savings.

Content Based Page Sharing (CBPS) is one of the most widely used memory deduplication techniques to improve memory efficiency, since CBPS can be performed transparently in the hypervisor layer and it doesn't require any mod-

VEE '14, March 1–2, 2014, Salt Lake City, Utah, USA.
Copyright © 2014 ACM 978-1-4503-2764-0 /14/03…$15.00.
http://dx.doi.org/10.1145/2576195.2576204

ification to guest operating systems (OS). In CBPS, a memory scanner is adopted to scan memory pages of guest VMs periodically and to detect all identical pages that have same content, these identical pages can then be shared into a single physical page[1], and the redundant memory pages can be reclaimed and free back to the hypervisor. Thus CBPS can efficiently reduce the memory footprint of guest VMs and provide good opportunity to increase the number of VMs collocated on a host server.

In this paper, we mainly focus on Kernel Samepage Merging (KSM) [6], which is a widely-used implementation of CBPS, and it has been integrated into the Linux kernel archive [3]. In KSM, there are two global red-black comparison trees for the whole memory pages of a hosting server, named **stable tree** and **unstable tree** respectively. The stable tree contains already shared pages with write-protected, and the unstable tree contains only pages that are not shared yet (without write-protected). In each scan round, each candidate page needs to be compared with pages already in these two large global trees to detect page sharing opportunities. We define the term **futile comparison** as the page content comparison of a candidate page with other pages (both in the stable tree and unstable tree), which fails to find any page with the same content. Since KSM only maintains two global comparison trees for the whole memory pages, each global tree will contain a large number of nodes (or pages), e.g. 1M nodes (4KB page) for 4GB memory. For each candidate page, it needs to be compared with a large number of uncorrelated pages in the global trees, thus it will result in massive futile page comparisons and heavy overhead. And as the page scan performs repeatedly, the number of futile page comparisons will increase proportionally.

In this paper, we propose a lightweight page Classification-based Memory Deduplication approach named **CMD** to reduce futile comparison overhead meanwhile to detect page sharing opportunities efficiently. In CMD, pages are grouped into different classifications based on page access characteristics. Pages with similar access characteristics are suggested to have higher possibility with same content, thus they are grouped into the same classification. And the large global comparison trees are divided into multiple small trees that there are dedicated local comparison trees in each page classification. For each candidate page, its classification is firstly determined based on its access characteristics (e.g. write access count, write access distribution of sub-pages), and then it is searched and compared with pages only in the local comparison trees of its classification. Thus page comparisons are performed just in the same classification, and pages from different classifications are never compared, since they probably result in futile comparisons. In this paper, we mainly focus on page write access, which will modify page content and thus affect page sharing opportunities. We monitor write access of pages with a hardware-snooping based memory trace monitoring system, which is able to capture all memory write references with fine cache-block granularity, thus we can use not only the count of write accesses for each page (coarse granularity), but also write access distribution of sub-pages (fine granularity), which will be a better hint for page classification. Additionally, hardware-assisted page access monitor introduces negligible overhead, thus our implementation of CMD is lightweight and efficient.

Overall, we have made the following contributions:

- We perform a detailed profiling of KSM, we find that page content comparisons contribute a certain portion of the whole KSM run-time (about 44%). And futile comparisons contribute most of the page comparison overhead (about 83%).
- To reduce futile comparison overhead meanwhile to detect page sharing opportunities efficiently, we propose a lightweight page Classification-based Memory Deduplication approach named **CMD**. In CMD, pages are grouped into different classifications based on page access characteristics, the large global comparison trees are divided into multiple trees with dedicated local ones in each classification. Page comparisons are performed just in the same classification, and pages from different classifications are never compared (since they probably result in futile comparisons).
- We implement the CMD in our real experimental system. The experimental results show that, compared with the baseline KSM, the CMD can efficiently detect page sharing opportunities (by more than 98%) meanwhile reduce the number of page comparisons (by about 68.5%), and the futile comparison rate is also reduced by about 12.15% on average.

The rest of the paper is organized as follows: Section 2 introduces the background and motivation. Section 3 describes the design and implementation of page classification based memory deduplication. We describe the experimental methodology in Section 4 and demonstrate experimental results and discussion in Section 5. Related work and conclusion are in Section 6 and Section 7 respectively.

2. Background and Motivation

2.1 Kernel Samepage Merging

KSM (Kernel Samepage Merging) [6] is a scanning based implementation of CBPS, which detects and shares pages with same content into a single copy. Nowadays KSM is implemented as a kernel thread, which periodically scans memory pages of guest virtual machines co-running on a hosting server. Each candidate page is compared content with pages already in the comparison trees, and then it is inserted into the global trees based on comparison result.

[1] With all identical guest physical pages pointing back to the same machine physical page in the hypervisor's page table.

If multiple pages with same content are detected during a scan round, one of the pages is selected as the KSM page (or identical page), then other duplicate pages are merged and shared with the single KSM page, which is implemented by replacing the page table entries of duplicate pages to map to the KSM page, and the original space of duplicate pages are reclaimed and saved. Duplicated Pages are shared with Copy-On-Write (COW) mechanism, which means that all page table entries of duplicate pages are mapped with read-only permission. If a VM attempts to write a shared page, a page fault named COW fault will be triggered. The hypervisor handles the fault by allocating a new page frame and making a private copy of the page for the requesting VM, then the VM writes data in this new private page.

Nowadays, KSM maintains only two global red-black comparison trees for the whole memory pages of a hosting server, as shown in Figure 1: the stable tree and the unstable tree. Nodes of these two global trees are indexed directly with the content of pages (as key of red-black nodes). The stable tree maintains search structure for shared pages, while the unstable tree maintains only pages that are not yet shared. In each scan round, a candidate page is firstly compared with pages in the stable tree. If there is a match with an existed KSM page, the candidate page will be merged and shared with this matched KSM page. Otherwise, it needs to be further searched in the unstable tree: If there is a match in the unstable tree, the matched page will be removed from the unstable tree, then it will be merged with the candidate page and migrated into the stable tree with write-protected; if no match is found, the candidate page is inserted into the unstable tree (as a leaf node). In each full scan round, the unstable tree needs to be reconstructed, which means that pages in the unstable tree need to be re-inserted into the unstable tree to get the correct node position. This is because that pages in the unstable tree are not write-protected, and content of these pages might be modified during the scan round. As the increasing capacity of main memory, the size of these two global trees expands proportionally. Since KSM does not take any page access characteristics into account, for a candidate page, it needs to be compared content with a large number of uncorrelated pages which will induce massive futile comparisons.

There is a tradeoff of controlling the page scan speed of KSM. Fast scan can detect more page sharing opportunities, especially for short-lived page-sharing, however it will induce heavy CPU overhead due to frequent page content comparisons. Slow scan, on the other side, might lose some short-lived page sharing opportunities, but the CPU overhead is relatively lightweight. In KSM, the kernel thread scans pages of VMs in a batch-by-batch manner: all candidate pages are firstly separated into batches with each batch having the same number of pages; after scanning all pages in a batch, the KSM thread goes to sleep for a specified time; then it continues to scan pages in the next batch. The scan speed can be controlled by configuring the number of pages in a batch (it is 100 by default) and the sleep time between batches (20ms by default). Before a page is searched in the unstable tree, the checksum of it is re-calculated and then compared with its last checksum. The checksum serves as a filter of page comparisons in the unstable tree: if the checksum remains the same, it is a candidate page, and the page can be searched and compared with pages in the unstable tree; otherwise, it is considered as a volatile page and it will not be searched in the unstable tree. The computation of page checksum also induces some CPU overhead to the system (as shown in Figure 2).

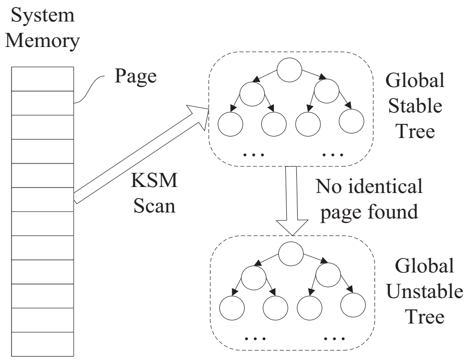


Figure 1. The global stable tree and global unstable tree in KSM. A candidate page is firstly searched and compared with pages in the stable tree. If no match is found, it needs to be further searched and compared with pages in the unstable tree.

2.2 Profiling of KSM

Table 1. KSM characteristics with different Configurations

Configuration	C0	C1	C2	C3	C4	C5
Pages to scan in each batch	100	200	400	800	1600	3200
Full scan time (seconds)	400	200	100	50	25	12.5

Table 1 shows the KSM characteristics with different configurations in our experimental system with 8GB memory (please refer to Section 4 for detailed system configuration). In KSM, there are two parameters that control scan-speed which can affect KSM sharing efficiency and run-time overhead: *pages_to_scan* and *sleep_millisecs*, which represents the number of pages to be scanned in a batch before the KSM thread goes to sleep, and how many milliseconds of time the KSM thread will sleep before turning to scan pages of the next batch of respectively. We adopt the default value of *sleep_millisecs* as 20ms and vary *pages_to_scan* from 100 (default, as C0) to 3200 (as C5). For our experimental system with 8GB physical memory, the corresponding of a full

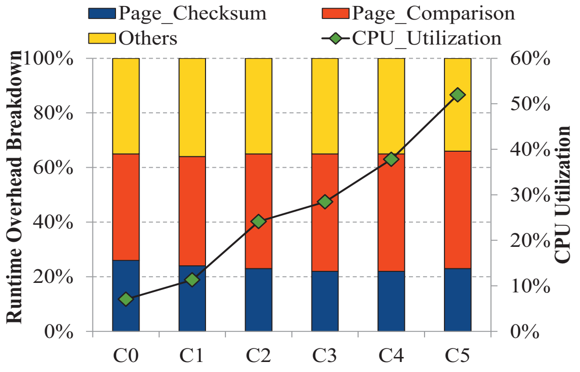


Figure 2. KSM run-time overhead breakdown and the CPU Utilization of the KSM thread as the number of pages to be scanned in a batch varied from 100 to 3200 with four guest VMs collocated on the host server.

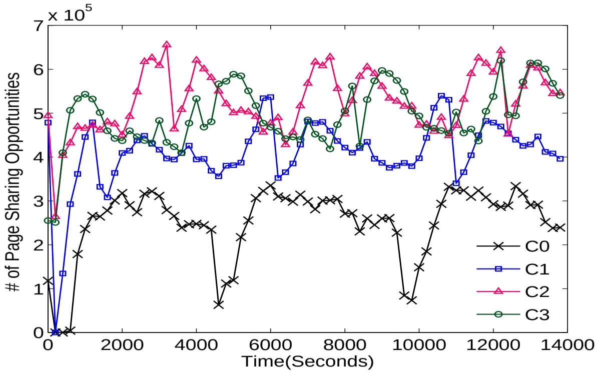


Figure 3. The number of page sharing opportunities with 4 VMs as the number of pages to scan in a batch varied from 100 (C0) to 800 (C3).

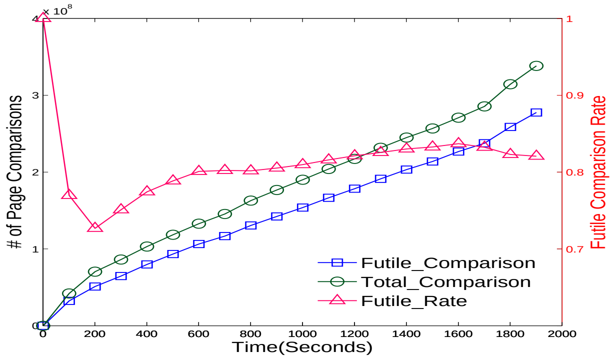


Figure 4. The number of total page comparisons and futile comparisons as the KSM thread running with 4 VMs, where KSM is configured to scan 400 pages and then sleep 20 milliseconds (default) in each batch.

scan time is varied from 400 seconds to 12.5 seconds respectively.

Figure 2 shows the KSM run-time overhead breakdown and the CPU Utilization of the KSM thread as the number of pages to be scanned in a batch varied from 100 (C0) to 3200 (C5) with four guest VMs collocated on the host server. We can see that as the number of pages scanned in a batch increasing, the CPU utilization increase correspondingly, it is about 8.52% for C1, it increase up to 24.13% for C3 and 53.09% for C5. The induced CPU overhead will degrade the performance of VMs, although it can find more page sharing opportunities (as shown in Figure 3). The KSM runtime overhead can be divided into 3 parts: *page checksum*, *page comparison*, and *others*. *Page checksum overhead* represents the time spending on calculating checksum of candidate pages to check whether it is volatile; *page comparison overhead* represents the time spending on page content comparison in both the global stable tree and unstable tree, and *others* represent the other KSM run-time overhead, such as inserting nodes in trees or removing nodes from trees, merging pages to be shared with COW, breaking COW when a shared page is modified. We can see that the overhead of *page checksum* and *page comparison* portions remain nearly the same (they don't not change with the scan speed). The *page comparison* contributes about 44% and the *page checksum* contributes about 22% of the total KSM run-time, while *others* contributes about 34%.

Figure 3 shows the number of page sharing opportunities with 4 VMs as the number of pages to be scanned in a batch varied from 100 (C0) to 800 (C3). The page sharing opportunities are varied with time as the phases of VMs changing. We can see that, the KSM can detect more page sharing opportunities as scan-speed increasing, since it can detect more short-lived page sharing in time. Normalized to C0, it can detect about 1.63x of page sharing opportunities for C1, about 2.08x for C2, and about 2.088x for C3 respectively. We can also see that, with 4 VMs running on our experimental system, C2 with 400 pages scanned in a batch is fast enough to detect almost all potential page sharing opportunities. Thus in the latter of this paper, without particularly pointed out, we will adopt C2 as our default KSM configuration.

Figure 4 shows the number of total page comparisons and futile page comparisons as the KSM thread running with 4 VMs. We can see that, as the KSM thread scans and compares pages periodically, the total number of page comparisons increases linearly. We can also see that although periodically repeated page comparisons can detect some additional page sharing opportunities, most of them are futile comparisons, which also increase almost linearly along with page scans periodically. We define the term **Futile_Rate** as follows:

$$Futile\_Rate = Futile\_PC\_Num / Total\_PC\_Num \quad (1)$$

Where *Futile_PC_Num* represents the number of futile page comparisons (failed to find sharing page), and *Total_PC_Num* represents the total number of page comparisons as periodically scans.

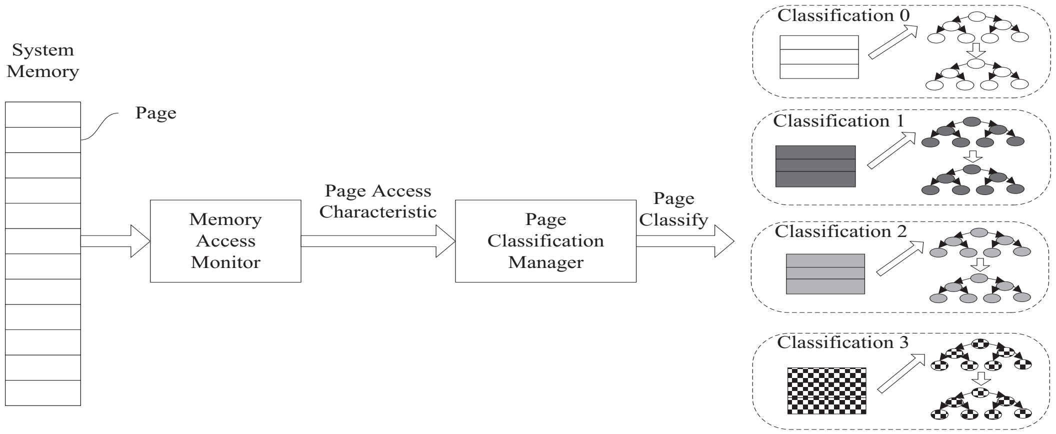


Figure 5. The page classification based memory deduplication implemented with KSM.

We can see that the **Futile_Rate** is finally become steady at about 83.64%. Thus we can conclude that: since nowadays the KSM compares pages with two large global trees and it does not take page access characteristics into account, most of the page comparisons are futile and do nothing help to find page sharing opportunities, thus it will induce heavy CPU overhead (about 44% as shown in Figure 2). And our goal of this paper is to reduce futile page comparisons overhead meanwhile detect page sharing opportunities efficiently in KSM page scans.

3. Classification-Based Memory Deduplication

In this section, we firstly introduce the overview of page Classification-based Memory Deduplication in subsection 3.1. Then we introduce a lightweight hardware-assisted approach to monitor page access characteristics in subsection 3.2. Finally, we introduce three different page classification approaches and analyze their Pros and Cons in subsection 3.3.

3.1 Overview

Since the KSM simply maintains two global comparison trees for all memory pages of a hosting server. To detect page sharing opportunities, each candidate page needs to be compared with a large number of uncorrelated pages in the global trees repeatedly, which will induce massive futile comparisons. The key innovation to reduce futile comparison is to break the global comparison trees into multiple small trees (with less page nodes in each one). Pages are grouped into multiple classifications, with dedicated local comparison tree in each page classification. A candidate page needs only to be compared with pages in its local comparison tree of its classification, which contains less page nodes. But the pages in its local tree are having much higher probability to have same content with the candidate page, thus it can reduce futile comparisons meanwhile detect page sharing opportunities efficiently.

There are two requirements for page classification approaches: (1) pages with high probability to have same content should be grouped into the same classification, thus the scan thread can detect page sharing opportunities in its local classification tree, as efficient as scanning pages in the global trees. And pages with low probability to have same content should be separated into different classifications, thus they will never be compared with each other. This can reduce futile page comparisons. (2) The page classification approach needs to be balanced, which means that the number of page nodes in each page classification should be nearly the same. Unbalanced page classification will result in a few of page classifications having large number of page nodes, and page comparisons in them will still induce massive futile comparisons as in the global comparison trees.

Figure 5 shows the page classification based memory deduplication named CMD. In CMD, there is a memory access monitor to capture all memory accesses to pages, especially write memory accesses, since write accesses modify pages content and thus affect page sharing opportunities. The memory access monitor maintains page access characteristics for each page: once a write access to a page is captured, it updates the page access characteristics of the corresponding page (e.g. write count). The page classification manager is responsible to group pages into different classifications based on page access characteristics: pages with similar access characteristics are grouped into the same classification. The page classification are performed in each scan round, which means that the access characteristics of pages captured during the last scan round are used to guide page classification in this scan round. And the memory access monitor continues to capture access characteristics of pages during this scan round, which will be used in the next scan round.

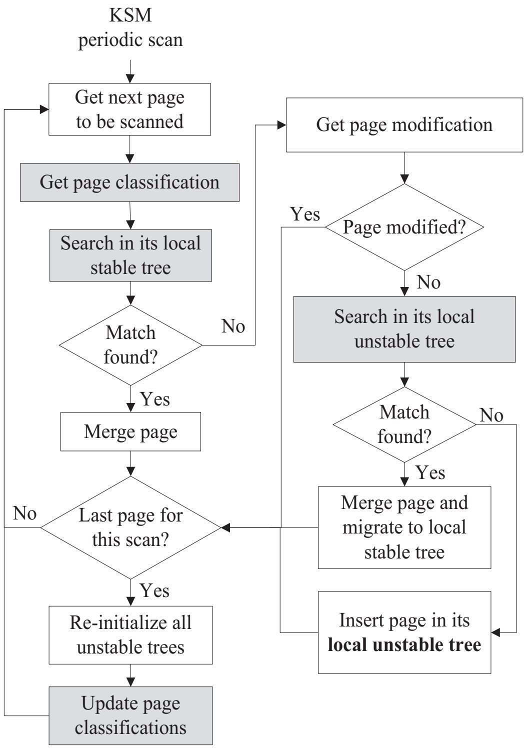

KSM periodic scan
Get next page to be scanned
Get page classification
Search in its local stable tree
Match found?
No
Yes
Merge page
Last page for this scan?
No
Yes
Re-initialize all unstable trees
Update page classifications
Get page modification
Page modified?
Yes
No
Search in its local unstable tree
Match found?
No
Yes
Merge page and migrate to local stable tree
Insert page in its local unstable tree

Figure 6. The Classification based KSM Tree algorithm flowchart.

After pages are grouped into classifications, there is a dedicated stable tree and a dedicated unstable tree in each page classification. In each scan, pages searching and comparison are performed in a classification-by-classification manner. In each page classification, a candidate page is firstly compared with pages in the local stable tree of its classification, if no match is found, the candidate page needs to be further compared with pages in the local unstable tree. When searching and comparison of all candidate pages in a classification finished, it starts to scan pages in the next classification. After finishing all pages scanning in all classifications, the the KSM thread will start the next scan round.

Figure 6 shows the classification based KSM tree algorithm flowchart. In each KSM periodic scan, the KSM kernel thread gets next page to be scanned, it firstly gets page classification from page classification manager and then searches in its local stable tree. If a match is found, the candidate page is merged with its identical page into its local stable tree. Otherwise, it gets page modification information (by comparing new calculated checksum with last checksum of the page), if the page is modified during last scan, it is considered to be volatile, and it will not be searched in the local unstable tree; if it is not modified, it is searched and compared with pages in its local unstable tree. If a match is found, they are merged into a shared page and migrated to its local stable tree; if no match is found, the candidate page is inserted into its local unstable tree. After finishing all pages searching in each scan round, all unstable trees in all page classifications need to be destroyed and re-constructed in the next scan round. Pages classifications also need to be updated based on page access characteristics which is monitored in this scan round.

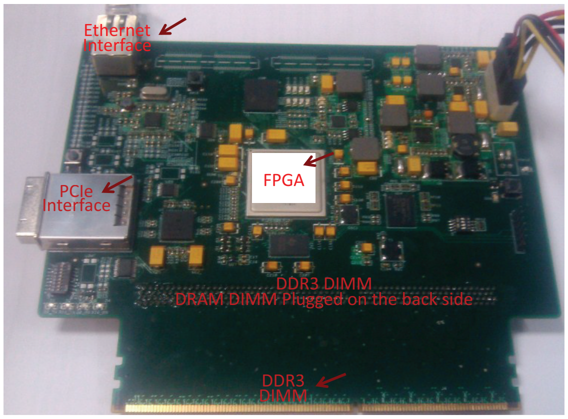


Figure 7. The HMTT board.

3.2 Page Access Monitor

In this paper, we adopt a hybrid hardware/software memory trace monitoring system named HMTT [7], which is based on hardware snooping technology. HMTT is DDR3 SDRAM compatible[2], and it is able to monitor full-system memory reference traces, including OS, VMMs, libraries, applications, and it has also been enhanced to support fine-grained objects [13], locks [18]. Figure 7 shows the HMTT board. It acts as a DIMM adaptor between the motherboard DIMM and the DRAM DIMM[3], thus it can monitor all memory transaction signals (issued to DRAM devices plugged on the HMTT board) on the DDR3 command bus. Then the corresponding memory read/write access references can be reconstructed based on interpreting DDR3 protocol with the FPGA on the HMTT board, thus HMTT can capture all memory traces at cache block granularity (since processors access DRAM memory when last level cache miss) on a real system. We have also implemented a PCIe interface on the HMTT to send memory traces to another machine with high bandwidth, thus the HMTT can support monitoring workloads with long-time running (e.g. hours). For more implementation detail, please visit the HMTT home page at `http://asg.ict.ac.cn/hmtt/`.

In this work, we don't need to get detailed memory access traces, instead, we just need write access characteristics of pages, such as write access count, write distribution of sub-pages. Thus we can directly maintain the access information of pages with a SDRAM buffer on the HMTT board. When a write access is monitored, it gets the physical page

[2] In this version, it can work with DDR3-800MHz.

[3] The HMTT is plugged directly on the motherboard DIMM, and the DRAM DIMM is plugged on the HMTT board.

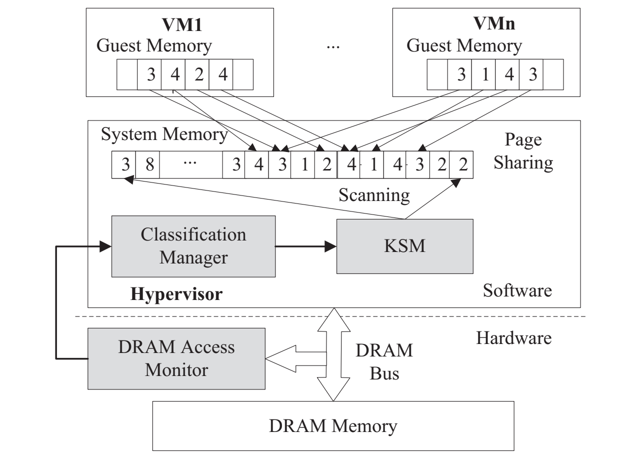


Figure 8. The overview of partitioned based KSM.

frame number (pfn) from request address, then it updates the access characteristics of the page. Furthermore, we have also implemented a Gigabit Ethernet interface on the HMTT as shown in Figure 7, and the page access information is sent periodically back to the experimental server with this Ethernet interface (the time interval is smaller than the KSM scan period). In software, we have implemented a shared memory buffer between the receiving process and the kernel (as a kernel module), thus the KSM thread can get the feedback page access characteristics periodically, and this feedback information is used to guide page classification. Since the HMTT adopts hardware snooping technology, monitoring page access characteristics introduces negligible overhead to the hosting server. And furthermore, HMTT can capture memory accesses at fine granularity (e.g. cache block), thus we can get more detailed and fine-grained page access characteristics, such as write access distribution among sub-pages. In contemporary processors, there is only an access bit and a dirty bit for each page can be used to indicate whether it is accessed or written recently. Although HMTT is an assistant device that can not be deployed in every server node, the main logic for memory access monitoring and counting can be easily integrated into on-chip memory controller. Here we only evaluate the benefit of such a mechanism while leaving the architecture issue for future research.

3.3 Page Classification

Page classification algorithm is critical to reduce futile page comparisons meanwhile to detect page sharing opportunities efficiently. In this paper, we consider the following three simple page classification algorithms:

- **CMD_Address**: pages are statically classified by their physical addresses (or page frame number). For example, in our experimental system with 8GB physical memory, pages are grouped into 8 classifications like that: pages with address from 0B to 1GB are classified into classification 0, pages with address from 1GB to 2GB are classified into classification 1, and so on. CMD_Address approach is probably to group pages into different partitions in balance, thus it can reduce page comparisons within each classification. However, it might lose some page sharing opportunities since it does not take page access characteristics into account, pages with the same content might be placed into different classifications, and they have no chance to be compared and shared with each other.

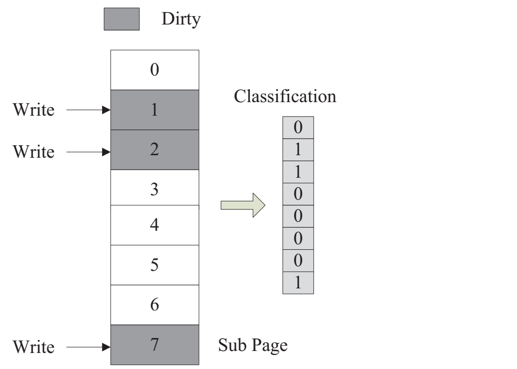


Figure 9. An example of CMD_Subpage_Distribution page classification.

- **CMD_PageCount**: the memory access monitor captures count of write accesses for each page during each scan round, and pages are simply partitioned by their write access count. For example, when we set the write access threshold as 8, then pages with the count of write accesses from 0 to 8 are placed into classification 0, and pages with the count from 8 to 16 are placed into classification 1, and so on. CMD_PageCount considers access characteristics as a whole page, thus it can improve classification accuracy rate and detect more page sharing opportunities.

- **CMD_Subpage_Distribution**: each page is divided into multiple sub-pages (e.g. 4 1KB sub-pages), and the memory access monitor maintains write access characteristics for all sub-pages. Pages with the same sub-page access distribution are grouped into the same classification. Figure 9 shows an example of page classification based on sub-page distribution. A page is divided into 8 sub-pages, each sub-page is with a dirty bit to indicate whether it is written (or modified) during a scan round. In this example, the page has only three sub-pages of 1,2,7 being written, thus the classification of this page is (0,1,1,0,0,0,0,1). With fine grained sub-page access characteristics, CMD_Subpage_Distribution has more opportunity to group pages with the same content into the same classification, thus it can detect page sharing opportunities more efficient.

In addition to the above page classification approach, we can make some optimizations to further reduce page comparison overhead. Pages that have not been modified during last scan round can be treated as a special classification. And they don't need to be searched and compared with each other, because the content of these pages have been compared and failed to find any sharing opportunities before. If the content of them are not modified, page comparisons in this classification will definitely result in futile page comparisons. Thus for this page classification, we can just put pages in a set without maintaining any comparison trees.

4. Experimental Methodology

We carried out our experiments with two 2.00GHz Intel Xeon E5504 processors with EPT enabled. Each E5504 processor has 4 physical cores and we have disabled the HyperThread. There are 3-level caches in each processor, the L1 instruction and data caches are 32KB each and the L2 cache is 256KB, both the L1 and L2 are private in each core. The L3 cache is 16-way 4MB and shared by all four cores in each processor. The cache block size is 64-Byte for all caches in the hierarchy. The total capacity of physical memory is 8GB with one dual-ranked of DDR3-800MHz. The host server runs CentOS-6.2 with Linux kernel 3.6.0. We implement CMD based on KSM of Linux 3.6.10. We use libpcap [5] to get ethernet packets with feedback page access characteristics from Ethernet interface of the HMTT board. We use QEMU [9] with KVM [2] (qemu-kvm-1.2.0) to support guest VMs. Each guest VM is configured with 1 virtual CPU and 2GB main memory, we boot 4 VMs in parallel as our default configuration. The guest VMs are running 64-bit CentOS-6.3 with Linux kernel 2.6.32-279. We choose to run the following workloads inside guest VMs:

- **Kernel Build**: we compile the Linux kernel 3.6.10 in guest VMs. We begin this benchmark after the VMs are fully booted and static sharing opportunities are detected.
- **Apache Server**: we run the ab [1] benchmark on Apache httpd server. We test a local web site in guest VMs with 24 of concurrency requests.
- **MySQL Database**: we run the SysBench [4] with MySQL database in guest VMs. We test database with 1-thread and the oltp-table-size is configured as 1500000.

The CMD configuration parameters are set as follows[4]. For CMD_Address, we separate the 8GB machine physical memory into 8 page classifications with 1GB memory in each page classification. For CMD_PageCount, we adopt the write access count threshold for each page of 64, and since we adopt 16 page classifications, all pages that with write accesses exceeding 1024 are placed into the last page classification. And for CMD_Subpage_Distribution, we adopt 4 sub-pages access distribution (with 1KB in each sub-page) to guide 16 page classifications. And to get the page classification as sub-page dirty map, we adopt a threshold of 16 for the number of write access to each sub-page, which means that when the number of write access for a sub-page exceeds 16, the dirty bit of this sub-page is set to 1; otherwise it is set to 0.

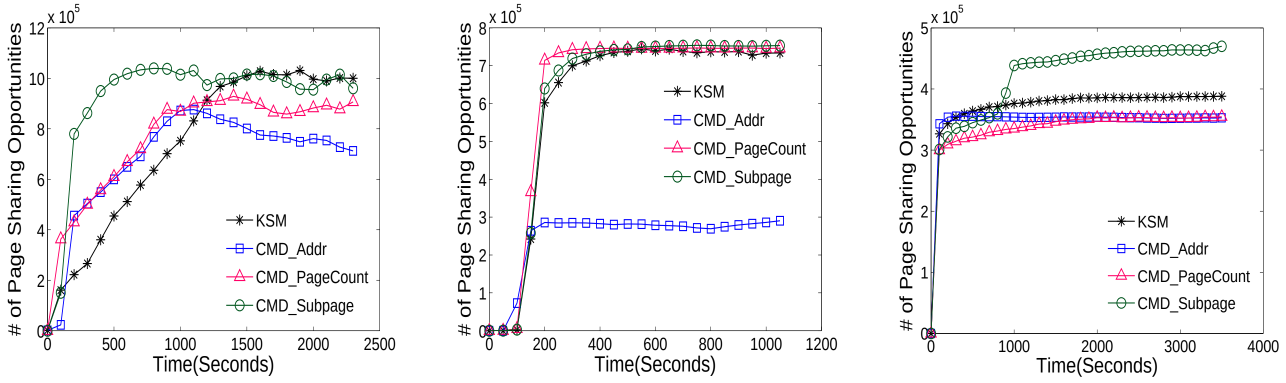
5. Experimental Results

Figure 10 shows the page sharing opportunities of different workloads with 4VMs. For *Kernel Build* workload in Figure 10(a), we can see that the KSM can detect the most page sharing opportunities (it is about 1.02E6 sharable pages). But since it maintains pages into large global comparison trees, candidate pages needs to be compared with a large number of uncorrelated page nodes, thus it takes a little longer time to reach its maximum page-sharing state. On the other hand, The CMD_Addr detects the least page sharing opportunities, it is only about 71% of the KSM (about 7.25E5). That is because the CMD_Addr groups pages into different classifications simply based on physical page frame number, but without taking any page access characteristics (which affect page content) into account. Pages with same content but with long-distance page frame number will be separated into different page classifications. And since this classification approach is static, these pages have no chance to be detected and shared with each other. The CMD_PageCount can detect about 87.1% (about 8.89E5) page sharing opportunities as the KSM, this is because it adopts coarse granularity page access characteristics (the count of write accesses in each scan round) to guide page classification. It can achieve higher accuracy, however there is still some room for improvement. Finally the CMD_Subpage is able to detect nearly the same page sharing opportunities (about 1.00E6), which is about 98% of the KSM. This result proves that fine-granularity sub-page write access distribution is a better guide for page classification. We can also see that the CMD_Subpage can detect page sharing opportunities more quickly than the KSM, since each candidate page costs less time overhead to be scanned and compared in its local comparison trees.

For *Apache* workload as shown in Figure 10(b), we can see that the KSM can detect almost the same page sharing opportunities (about 7.45E5) with the CMD_PageCount and the CMD_Subpage approach. However the CMD_Addr performs quite poor, it can detects about only 38.9% page sharing opportunities of the KSM (about 2.9E5). This further indicates that static page classification has poor adaptivity, and it needs to adopt dynamic page classification approach. In this paper we prove that fine granularity dynamic page access characteristics is a good hint for page classification, since it has a close relationship with page content and thus page sharing opportunities.

For *MySQL* workload as shown in Figure 10(c), we can see that the CMD_Addr and the CMD_PageCount detects almost the same page sharing opportunities (about 3.54E5),

[4] However they are not the best configurations, since we just choose them based on some coarse parameters searching.

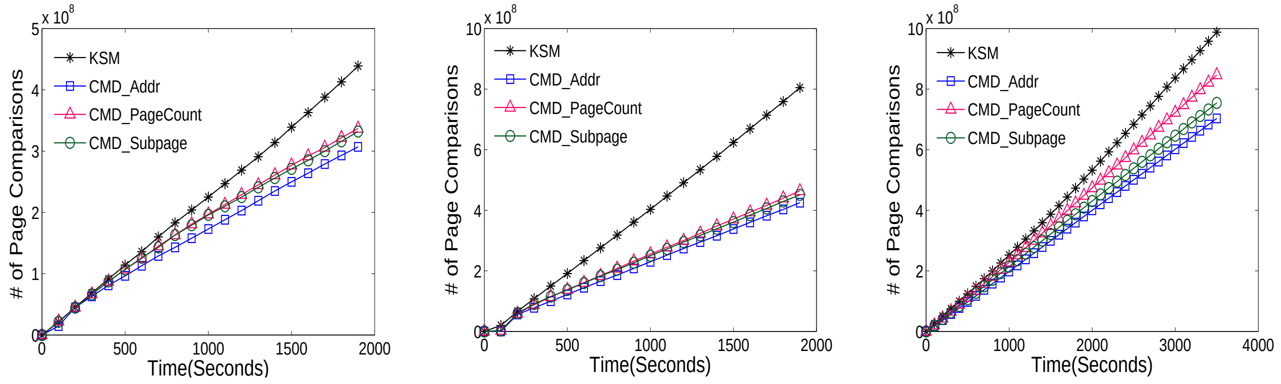


(a) Kernel Build

(b) Apache

(c) MySQL

Figure 10. The page sharing opportunities with 4 VMs.



(a) Kernel Build

(b) Apache

(c) MySQL

Figure 11. The number of page comparisons of different workloads with 4VMs.

which is about 91.5% of the KSM (about 3.87E5). It is worth noting that the CMD_Subpage can detect even more page sharing opportunities (about 4.69E5), which is about 1.21x of the KSM. The probable reason for it is that, in MySQL workload, there exists abundant short-lived page sharing opportunities. However the KSM is failed to detect them since it costs longer comparison time for each candidate page which inducing massive futile comparisons. They can be detected by the CMD_Subpage approach, because for each candidate page, it can be scanned more quickly just in its local comparison tree, which makes it detect short-lived page sharing opportunity more efficient.

Figure 11 shows the number of pages comparisons of different workloads with 4 VMs. For *Kernel Build* workload as shown in Figure 11(a), we can see that the KSM with large global comparison trees induces the most number of page comparisons. While the CMD_Addr has the least number of page comparisons, it is about 56.8% of the KSM, since CMD_addr has the best balanced page classifications (with address). The CMD_PageCount and the CMD_Subpage have nearly the same number of page comparisons, it is about 73.3% of the KSM. Although the CMD_Subpage induces about 1.08x of page comparisons than the CMD_Addr, it can detect page sharing opportunities more efficient (as shown in Figure 10). For *Apache* workload as shown in Figure 11(b), the KSM also has the most number of page comparisons, and the CMD_Addr has the least number of page comparisons which is about 55.5% of the KSM. The CMD_PageCount and the CMD_Subpage have almost the same page comparisons, which is about 55.7% of the KSM. For *MySQL* workload as shown in Figure 11(c), The CMD_Addr also has the least number page comparisons, which is about 71.2% of the KSM. While the CMD_Subpage is about 76.6% of the KSM, this is less than the CMD_PageCount, which is about 86% of the KSM. Thus we can conclude that the CMD_Subpage is the best tradeoff between detecting page sharing opportunities and reducing page comparisons (especially futile comparisons).

Figure 12 shows the percentage of futile rate reduction with 4 VMs, where the baseline is with the KSM approach. We can see that all these three approaches can reduce futile rate. On average it can reduce at about 4.77%,

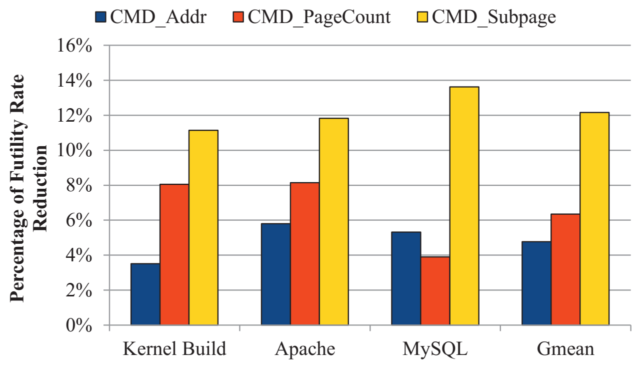


Figure 12. The percentage of futile rate reduction with 4 VMs, where the baseline is with the KSM approach.

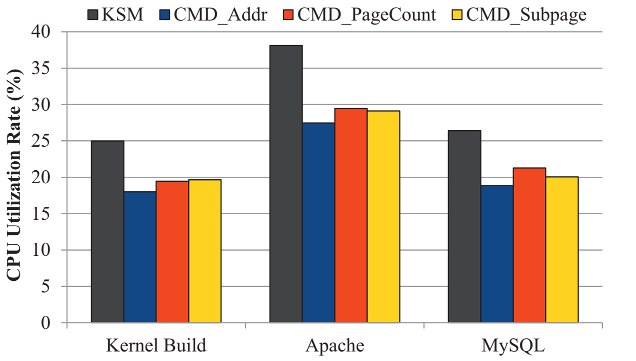


Figure 13. The average CPU Utilization Rate of the KSM kernel thread for different approaches. The CPU Utilization is got from top measurements taken every second.

6.35% and 12.15% for the CMD_addr, CMD_PageCount and CMD_Subpage respectively. The maximum reduction of the CMD_Subpage approach is about 13.62% for *MySQL* workload. This result prove the accuracy of the CMD_Subpage approach for page classification, which is based on fine granularity sub-page access distribution characteristics.

Figure 13 shows the average CPU Utilization Rate of the KSM kernel thread for different approaches. We get the CPU Utilization from top measurements taken every second. We can see that all the other three CMD approaches (dividing the global comparison tree into multiple comparison trees) can reduce the CPU overhead compared with the KSM. For *Kernel Build* workload, the CPU Utilization Rate is about 24.98% for the KSM, which is reduce to 18.00%, 19.45% and 19.65% for the CMD_Addr, CMD_PageCount and CMD_Subpage respectively. This is main because that the CMD approaches can reduce the number of futile page comparisons and reduce the CPU run-time overhead. For the *Apache* workload, the CPU Utilization Rate is about 38.12% for the KSM, and it reduces to 27.48%, 29.44% and 29.12% for the CMD_Addr, CMD_PageCount and CMD_Subpage respectively. For the *MySQL* workload, the CPU Utilization Rate is about 26.40% for the KSM, and it reduces to 18.86%, 21.28% and 20.05% for the CMD_Addr, CMD_PageCount and CMD_Subpage respectively.

6. Related Work

Limited main memory size has become one of the major bottlenecks in virtualization environment, and as an efficient approach to reduce server memory requirement, memory deduplication thus has attracted a large body of work on it. Disco virtual machine monitor (VMM) [10] was one of the first systems that implemented page sharing technique, however it needed to modify the guest OS to explicitly track page changes to build knowledge of identical pages. The content-based page sharing (CBPS) technique was firstly exemplified by VMWare's ESX server [29], in which CBPS required no assistance from the guest OS and it was performed transparently in the hypervisor layer. Thus CBPS had become the most widely used page sharing technique both in industry and academia. Kloster et al. [20] designed and implemented content based page sharing in the Xen hypervisor. They found that the unified disk caches were often the most significant cause of redundancy in virtualized systems.

KSM (Kernel Samepage Merging) [6] is a scanning based mechanism to detect page sharing opportunities. KSM is implemented as a kernel thread, which periodically scans pages of guest VMs to detect identical pages (based on page content comparison). Chang et al. [12] conducted empirical study on the effectiveness of KSM for various kind of workloads. They found KSM achieved effective memory sharing which could reduce memory usage by around 50% for I/O-bound applications. And they also found KSM would cause higher run-time overhead for CPU-bound applications caused by futile page comparisons. Rachamalla et al. [24] studied the trade-off between KSM performance and CPU overhead with different KSM configurations, then they developed an adaptive scheme to maximize sharing opportunities at minimal overhead. Barker et al. [8] proposed an empirical study of memory sharing in virtual machines through an exploration and analysis of memory traces captured from real user machines and controlled virtual machines, they found that sharing tended to be significantly more modest, and the self-sharing contributed a significant majority of the total sharing potential. Yang et al. [32] evaluated memory overcommit features of memory sharing and swapping in Xen, they also proposed an adaptive memory overcommit policy to reach higher VM density without sacrificing performance.

Gupta et al. proposed Difference Engine [17] to firstly support sub-page sharing, in which full memory pages were broken down into multiple sub-pages and memory sharing was performed at the fine sub-page granularity. They further proposed patching technique which would store similar pages by constructing patches, and worked together with compression of non-shared pages to reduce the memory footprint. With these optimizations, it could save memory up to 90% between VMs running similar applications and operating systems and up to 65% even across VMs running disparate workloads. Wood et al. [30] proposed Memory Buddies which adopted intelligent VM collocation within a data center to aggressively exploit page sharing benefits. KSM++ [21] adopted I/O-based hints in the host to make the memory scanning process more efficient and to exploit short-lived sharing opportunities. Miller et al. [22] proposed a more effective memory deduplication scanner XLH, which could exploit sharing opportunities earlier. XLH generated page hints in the host's virtual file system layer, then moved them earlier into the merging stage. Thus XLH could find short-lived sharing opportunities. However, the XLH could only worked for virtual machines with intensive file, and it still adopted the KSM with global trees for the whole system memory. Chiang et al. [14] proposed a Generalized Memory de-duplication engine that leveraged the free memory pool information in guest VMs and treated the free memory pages as duplicates of an all-zero page to improve the efficiency of memory de-duplication.

Satori system [23] implemented sharing by watching for identical regions of memory when read from disk, it could find short-lived sharing opportunities effectively, however it required modifications to the guest operating systems. Sharma et al. [27] extended the page deduplication of KSM for page caches, they proposed Singleton to address the double-caching problem, which implemented an exclusive cache for the host and guest page cache hierarchy. Kim et al. [19] proposed a group-based memory deduplication scheme that allowed the hypervisor to run with multiple deduplication threads, each of which was in charge of its dedicated group. However their sharing group was aimed to provide performance isolation and secure protection among different groups. Deng et al. [15] also proposed a similar memory sharing mechanism based on user groups to support isolation and trustworthiness mechanism between different users on the same host. Both of the above work did not take futile page comparison overhead into account, while our goal of page classification in this paper is to reduce futile page comparison overhead based on page access characteristics. Sha et al. proposed SmartKSM [26] to divide memory footprints into several sets in KSM based on page types and process aspect. Sindelar et al. [28] proposed the design of graph models to capture page sharing across VMs. They also developed sharing-aware algorithms that could collocate VMs with similar page content on the same physical server. Xia and Dinda [31] argued that in virtualized large scale parallel systems, both intra- and inter- node memory content sharing was common, they then proposed an effective sharing memory detection system by using a distributed hash table. In IBM Active Memory Deduplication [11], the hypervisor created signatures for physical pages (in a deduplication table) to reduce full page content comparison overhead.

7. Conclusion

In this paper, we firstly perform a detailed profiling of the KSM run-time, and we find that there exists massive futile page comparisons, because the KSM thread scans on two large global comparison trees. We then propose a lightweight page Classification-based Memory Deduplication approach named CMD. In CMD, pages are divided into different classifications based on page access characteristics, and the large global comparison trees are divided into multiple trees with dedicated ones in each page classification. Page comparisons are performed just in the same classification, and pages from different classifications are never searched and compared with each other, since they are probably futile comparisons. We adopt a lightweight hardware assisted memory trace monitoring system to capture fine granularity page access characteristics. We implement CMD based on KSM in our real experimental system, and the experimental results show that the CMD can detect page sharing opportunities efficiently (more than 98% of the KSM), meanwhile reduce page comparisons and reduce the futile rate by about 12.15% on average with fine granularity subpage access distribution characteristics.

Acknowledgements

The authors thank the anonymous reviewers for their constructive suggestions. This work is partially supported by the National Natural Science Foundation of China (NSFC) under Grant Nos. 60925009, 61221062, 61331008, the National Basic Research Program of China (973 Program) under Grant No. 2011CB302502, the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDA06010401, and the Huawei Research Program under Grant No. YBCB2011030. Yungang Bao is partially supported by the CCF-Intel Young Faculty Research Program (YFRP) Grant.

References

[1] ab - apache http server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[2] Kvm-kernel based virtual machine. `http://www.linux-kvm.org/page/Main_Page`.

[3] Linux 2 6 32 - linux kernel newbies. `http://kernelnewbies.org/Linux_2_6_32`.

[4] Sysbench: a system performance benchmark. `http://sysbench.sourceforge.net/`.

[5] Tcpdump and libpcap. `http://www.tcpdump.org/`.

[6] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Proceedings of the Linux Symposium (OLS'09)*, pages 19–28, 2009.

[7] Y. Bao, M. Chen, Y. Ruan, L. Liu, J. Fan, Q. Yuan, B. Song, and J. Xu. Hmtt: a platform independent full-system memory trace monitoring system. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measure-*

ment and modeling of computer systems, SIGMETRICS '08, pages 229–240, 2008.

[8] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman. An empirical study of memory sharing in virtual machines. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 273–284, 2012.

[9] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–46, 2005.

[10] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, Nov. 1997.

[11] R. Ceron, R. Folco, B. Leitao, and H. Tsubamoto. Power systems memory deduplication. In *IBM Redbooks*, 2012. http://www.redbooks.ibm.com/abstracts/redp4827.html.

[12] C.-R. Chang, J.-J. Wu, and P. Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on*, pages 244–249, 2011.

[13] L. Chen, Z. Cui, Y. Bao, M. Chen, Y. Huang, and G. Tan. A lightweight hybrid hardware/software approach for object-relative memory profiling. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 46–57, 2012.

[14] J.-H. Chiang, H.-L. Li, and T.-c. Chiueh. Introspection-based memory de-duplication and migration. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '13, pages 51–62, 2013.

[15] Y. Deng, C. Hu, T. Wo, B. Li, and L. Cui. A memory deduplication approach based on group in virtualized environments. In *Proceedings of the 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, SOSE '13, pages 367–372.

[16] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(9):34–45, Sept. 1974.

[17] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. In *8th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'08, pages 309–322, 2008.

[18] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen. Halock: hardware-assisted lock contention detection in multithreaded applications. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 253–262, 2012.

[19] S. Kim, H. Kim, and J. Lee. Group-based memory deduplication for virtualized clouds. In *6th Workshop on Virtualization in High-Performance Cloud Computing*, VHPC 2011, pages 387–397, 2011.

[20] J. F. Kloster, J. Kristensen, and A. Mejlholm. Determining the use of interdomain shareable pages using kernel introspection. In *Tech. rep., Aalborg University, 2007.* http://mejlholm.org/uni/pdfs/dat7_introspection.pdf.

[21] K. Miller, F. Franz, T. Groeninger, M. Rittinghaus, M. Hillenbrand, and F. Bellosa. Ksm++: Using i/o-based hints to make memory-deduplication scanners more efficient. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12)*, 2012. http://www.dcs.gla.ac.uk/conferences/resolve12/papers/session3_paper2.pdf.

[22] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa. Xlh: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, USENIX ATC'13, pages 279–290, 2013.

[23] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: enlightened page sharing. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 1–14, 2009.

[24] S. Rachamalla, D. Mishra, and P. Kulkarni. All page sharing is equal, but some sharing is more equal than others. 2013. http://www.cse.iitb.ac.in/internal/techreports/reports/TR-CSE-2013-49.pdf.

[25] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38(5):39–47, 2005.

[26] S. Sha, J. Li, N. Li, W. Ju, L. Cui, and B. Li. Smartksm: A vmm-based memory deduplication scanner for virtual machines. http://act.buaa.edu.cn/lijx/pubs/sosp2013.smartksm.pdf.

[27] P. Sharma and P. Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 15–26, 2012.

[28] M. Sindelar, R. K. Sitaraman, and P. Shenoy. Sharing-aware algorithms for virtual machine colocation. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 367–378, 2011.

[29] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[30] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 31–40, 2009.

[31] L. Xia and P. A. Dinda. A case for tracking and exploiting inter-node and intra-node memory content sharing in virtualized large-scale parallel systems. In *Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date*, VTDC '12, pages 11–18, 2012.

[32] X. Yang, C. Ye, and Q. Lin. Evaluation and enhancement to memory sharing and swapping in xen 4.1. In *XenSubmitt 2011.* http://www-archive.xenproject.org/files/xensummit_santaclara11/aug3/3_XiaoweiY_Evaluation_and_Enhancement_to_Memory_Sharing_and_Swapping_in_Xen%204.1.pdf.