UKSM: Swift Memory Deduplication via Hierarchical and Adaptive Memory Region Distilling

Nai Xia and Chen Tian, *State Key Laboratory for Novel Software Technology, Nanjing University, China;* Yan Luo and Hang Liu, *Department of Electrical and Computer Engineering, University of Massachusetts Lowell, USA;* Xiaoliang Wang, *State Key Laboratory for Novel Software Technology, Nanjing University, China;*

https://www.usenix.org/conference/fast18/presentation/xia

This paper is included in the Proceedings of the 16th USENIX Conference on File and Storage Technologies.

February 12–15, 2018 • Oakland, CA, USA

ISBN 978-1-931971-42-3

UKSM: Swift Memory Deduplication via Hierarchical and Adaptive Memory Region Distilling

Nai Xia[†] *Chen Tian*[†] *Yan Luo*[‡] *Hang Liu*[‡] *Xiaoliang Wang*[†]
[†]*State Key Laboratory for Novel Software Technology, Nanjing University, China*
[‡]*Department of Electrical and Computer Engineering, University of Massachusetts Lowell, USA*
{*xianai, tianchen, waxili*}*@nju.edu.cn*, {*Yan_Luo, Hang_Liu*}*@uml.edu*

Abstract

In cloud computing, deduplication can reduce memory footprint by eliminating redundant pages. The responsiveness of a deduplication process to newly generated memory pages is critical. State-of-the-art Content Based Page Sharing (CBPS) approaches lack responsiveness as they equally scan every page while finding redundancies. We propose a new deduplication system UKSM, which prioritizes different memory regions to accelerate the deduplication process and minimize application penalty. With UKSM, memory regions are organized as a distilling hierarchy, where a region in a higher level receives more CPU cycles. UKSM adaptively promotes/demotes a region among levels according to the region's estimated deduplication benefit and penalty. UKSM further introduces an adaptive partial-page hashing scheme which adjusts a global page hashing strength parameter according to the global degree of page similarity. Experiments demonstrate that, with the same amount of CPU cycles in the same time envelop, UKSM can achieve up to 12.6$\times$ and 5$\times$ more memory saving than CBPS approaches on static and dynamic workloads, respectively.

1 Introduction

In cloud computing, multiple virtual machines (VMs)/ containers (*e.g.*, dockers)/ processes are consolidated to share a physical sever. For a public cloud, the more VMs that can be packed into one host, the more VMs can be sold to tenants. For a private cluster, the more processes that can be packed into one host, the fewer the number of hosts needs to be purchased and maintained. In this context, available memory space can be a major bottleneck which limits the number of VMs/containers/processes that can be consolidated [1].

Memory deduplication can reduce memory footprint by eliminating redundant pages. This is particularly true when similar OSes/applications/data are used across different VMs. For instance, Chang *et al.* [2] observed as much as 86% redundant pages in real-world applications. Essentially, memory deduplication detects those redundant pages, and merges them by enabling transparent page sharing. It is important to mention that deduplication has penalty besides benefit. These shared pages are managed in a copy-on-write (COW) fashion, that is, when a write request happens to one of the transparently shared pages, this specific page can not be shared any more. A new copy of this page will be generated in the memory so that the write request is applied there, which is called page *COW-broken*.

The responsiveness of the deduplication process to newly generated pages is critical. For a production system, the memory is always dynamic, where pages come and go. As demonstrated by our typical cloud computing workload experiment (Section 8), if an approach cannot catch up with the generation speed of memory redundancy, memory pages would be swapped out to the disk, and the whole system is slowed down.

State-of-the-art Content Based Page Sharing (CBPS) approaches lack responsiveness as they equally scan every page to find redundancies. CBPS is a major deduplication method in Linux, Xen and VMware [3, 4, 5]. It is capable of full memory scan and it is easy to be integrated into main stream systems. For example, Linux's Kernel Same-page Merging (KSM) is a kernel feature that deduplicates pages for both virtualized and non-virtualized environments. In short, CBPS uses a scanner to calculate the hash value for every candidate page. If two pages share the same hash value, a byte-by-byte memory comparison is performed. If duplication confirmed, one page is merged to the other. It should be noted that: *NOT* all pages are created equal. Due to their applications' nature, some pages have little chance of being identical to others. These so called *sparse* pages should be tested in the last place. Some pages, although identical to others at the very beginning, can quickly become either COW-broken or freed. We refer to them as *COW-broken* pages and *short-lived* pages respectively. An ideal candidate page for deduplication should remain static (*i.e.*, not COW-broken or freed) for a reasonable period of time. A deduplication approach should prioritize these *statically-duplicated* pages. Further, deduplication operations performed on different pages may have different degrees of performance impacts on applications. We should also minimize deduplication's penalty on running applications due to operations such as page table locks and recovery of COW-broken pages.

Our observation is that *pages within the same memory region present similar duplication patterns* (Section 3). Here a memory region refers to a continuous virtual memory region allocated by an application (*i.e.*, allocated by *malloc*, *brk*, *mmap* etc). In some regions, most pages are statically-duplicated. In other regions, most identical pages may quickly become COW-broken or freed. According to the dominant page pattern, we can label a region as one of the four types of *sparse*, *COW-broken*, *short-lived* and *statically-duplicated* (*i.e.*, with a high duplication ratio, long-lived and seldom-changed). Intuitively, if we can prioritize pages in statically-duplicated regions for testing redundancy, the deduplication speed could be significantly accelerated. However, the challenge is how to distill these regions without testing every page at the first place?

Our key insight is that for each memory region, we can estimate its duplication ratio by sampling only a portion of all pages, at the same time monitor its degree of dynamics and lifetime. We can then distinguish sparse, short-lived and frequently COW-broken regions from statically-duplicated regions. To this end, we propose a new deduplication system *Ultra KSM* (UKSM). Build on top of KSM, UKSM improves traditional CBPS designs by prioritizing statically-duplicated regions over other regions to accelerate the deduplication process and minimize application penalty (Section 4).

UKSM introduces a hierarchy of sampling levels, each of which maintains a linked-list of memory regions. Each time an application *mmap*-s a new memory region, this region is immediately inserted into the list of the bottom level, which has the lowest scanning speed hence the lowest sampling density. A single thread iterates over levels to sample and deduplicate pages in each level. After each round of sampling, the duplication ratio and COW ratio of each region are compared with a set of threshold values. Once a memory region is identified as a potential statically-duplicated region, it is promoted from the current level to the next higher level which has a higher scanning speed hence a higher sampling density. This hierarchical architecture ensures system responsiveness by investing more CPU resources in regions in higher levels (Section 5).

To minimize the computational cost, we further develop a new partial-page hashing scheme called Adaptive Partial Hashing (APH). Let *page hash strength* denotes the number of bytes hashed in each sampling page. We define *profit* as the time saved compared to the strongest page hash strength and *penalty* as the wasted time of futile memory comparison due to hash collision. APH adaptively selects a global page hash strength to maximize the overall benefit which is *profit* subtracting *penalty*. Our novel progressive hash algorithm can support hash strength adaptation with incremental cost. Note that APH can improve other deduplication approaches as well since they are mostly hash-based (Section 6).

UKSM is implemented in both Linux kernel and Xen. The approach can detect and merge duplicated memory pages in real-time without intruding other parts of a system (*e.g.*, I/O, file system, etc). Experiments demonstrate that, with the same amount of CPU cycles in the same time envelop, UKSM can achieve up to 12.6×/5× more memory saving than CBPS approaches (*e.g.*, KSM) on static/dynamic workloads, respectively. UKSM also significantly outperforms XLH (*i.e.*, 50% more memory saving with the same amount of CPU consumption), a state-of-the-art I/O hint based approach. UKSM introduces negligible CPU consumption (around 0.2% of one core) when the host has no more page to be deduplicated, at the same time can respond to emerging duplicated pages rapidly (Sections 7 and 8).

UKSM is an open source project and benefits a wide range of applications [6]. Its patches for Linux kernel were first released in 2012 and have been kept synchronized with upstream kernel releases ever since. UKSM has been downloaded for over 30,000 times (at our site [6] alone, not including those re-distributed by other developers) at the time of the paper's publication. Besides the default versions, UKSM was also ported to kernels for desktop/server Linux systems [7, 8, 9, 10, 11] and Android systems [12, 13, 14, 15, 16] by third-party developers.

2 Related Work

Content-based Page Sharing (CBPS) VMWare ESX server [5] is the pioneer of content based page sharing approaches, where memory pages are scanned one-by-one. To control realtime CPU overhead, pages are randomly selected at a fixed scanning speed. A hash function is applied to each page for checking the similarity among pages. Pages that hash to the same value are byte-by-byte fully compared before they can be shared through copy-on-write. IBM Active Memory Deduplication [17] uses a similar approach for hypervisors in Power systems. CBPS for Xen was proposed by Kloster et al. [4] and later extended by XDE [18]. They detect page similarity by SuperFastHashing 64-byte blocks at two fixed locations in each page [19].

Linux Kernel Same-page Merging (KSM) [20] allows applications (including KVM [21]) to share identical memory pages via full page comparison. KSM works well for deduplicating fairly static pages. Singleton [22] extends KSM to consider host disk cache in a VM environment and improves the scanner from full-page comparison to SuperFastHash-based hash comparison. Red Hat Enterprise Linux uses a dedicated user space daemon named *ksmtuned* [23] to adjust KSM scanning speed under certain circumstances. For example, it increases the scanning speed when memory usage exceeds some threshold and the system is starting virtual machines. It is a very limited approach that simply adjusts scanning speed according to coarse grained system information which may not always imply page duplication. KSM would waste CPU resources if this kind of implication fails. It is hard for *ksmtuned* to achieve maximum saving across different workload patterns [3], although it does improve performance if optimized case by case.

Instead of treating every page equally, UKSM prioritizes different memory regions to accelerate the deduplication process. APH shares partial page hashing ideas [18] but can adapt the global page hash strength according to page similarity in the whole system.

Catalyst [24] offloads page hashing computation to GPU to improve deduplication performance. The need of special hardware support increases deployment complexity. SmartMD [25] uses page access information monitored by lightweight schemes to improve the efficiency of large page (e.g. 2M-pages) deduplication. This work is orthogonal to UKSM since we address the more general problem of page deduplication.

I/O hint based page sharing KSM++ [26] proposed a deduplication scanner based on I/O hints. XLH [27] utilizes cross layer I/O hints in the host's virtual file system to find sharing opportunities earlier without raising the deduplication overhead. A generalized memory deduplication was proposed in [28] that leverages the free memory pool information in guest VMs. It treats free memory pages as duplicates of an all-zero page to improve the efficiency of deduplication. I/O-hinted approaches cannot detect dynamically created duplicated pages (*e.g.*, anonymous pages created by applications in Docker containers).

CMD [29] is a classification-based deduplication approach. Pages are classified according to their access characteristics. Comparison trees introduced in KSM are subsequently divided into multiple trees dedicated to each class. Thus, page comparisons are performed only in the same class which reduces futile comparison among different classes. However, the above strategies require dedicated *hardware* monitors to capture system I/O or page access characteristics, which incurs significant deployment complexity.

In this paper, we focus on improving CBPS because of its capability of full memory scan and easy integration to all existing systems, neither of which is the case for I/O hint based page sharing option.

Storage deduplication is different Deduplication projects in disk storage systems [30, 31, 32, 33, 34] are important related works. However, there exist two significant differences.

First, UKSM faces the challenge of responsiveness which is not the case for disk storage deduplication projects. For instance, when a large volume of duplicated pages are generated, memory deduplication system needs to quickly identify and remove these duplicates before they exhaust available physical memory and cause memory swap out.

Second, since memory is dynamically updated while disk storage is relatively static, memory deduplication pays attention to more characteristics than just a duplication ratio that is the centerpiece for disk deduplication As reflected in this work, UKSM also considers COW ratio and lifetime characteristics of memory regions.

3 Observations

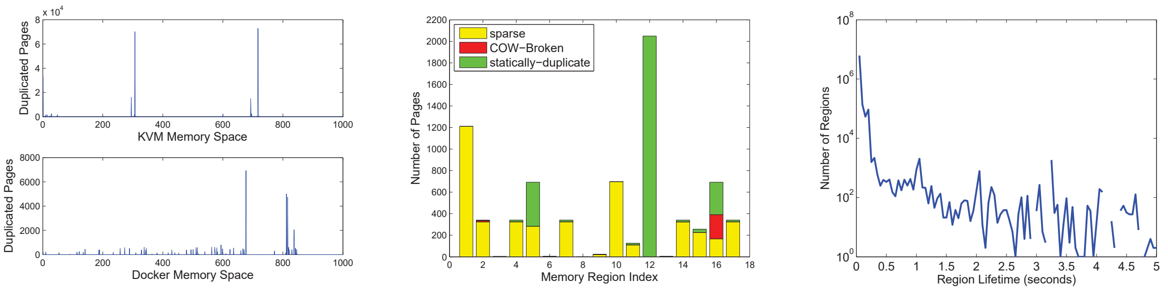
This section discusses two key observations that motivate the design of UKSM.

Observation # 1: Most pages within the same region present similar duplication patterns.

All heap memory allocation operations end up relying on *mmap* to claim memory spaces. For each call, mmap allocates a memory region that encompasses one or multiple virtual pages with continuous virtual addresses. Our intuition is that pages in the same memory region might exhibit same characteristics for deduplication. For instance, KVM exploits mmap to allocate memory space for each guest VM's OS. If two memory regions from different VMs store the same disk content for a long term, pages in them are friendly to deduplication. As a comparison, if a region of a network program serves as its busy network socket buffer, pages in it may not worth to be deduplicated even if many of them are identical. It will lead to frequent COW-broken operations.

Settings We use KVM and Docker as workloads for analysis of duplicated, COW-broken and short-lived pages. For the container workload, we make a Docker image from a Ubuntu based system with Apache web server and MySQL database serving a WordPress website. We then start three Docker containers from this image. For the KVM workload, we start three KVM virtual machines all installed with Ubuntu 16.04.

Results Page duplications demonstrate strong locality with respect to application memory regions. For both KVM and Docker, we evenly divide their virtual memory spaces (each contains many small memory regions) to 1,000 buckets. The number of duplicated pages in each bucket is presented in Figure 1(a). It is clear that most duplicated pages concentrate on a portion of memory



(a) Number of duplicated pages (b) A snapshot of Docker memory fragment (c) Distribution of short-lived regions

Figure 1: Memory regions in VM and container environments.

regions. We randomly demonstrate a bucket, which contains 18 memory regions from different processes, of one Docker's memory space. Note that some regions are so small in size (*e.g.*, regions 3, 6, 8) that they are almost invisible in the bar illustration. As shown in Figure 1(b), most pages in the same region share similarity in redundancy. Some regions are sparse and contain little duplications (*e.g.*, regions 1 and 2). Some are highly statically-duplicated (*e.g.*, region 12). Regions 5 and 16 are more complicated, where different kinds of pages coexist in the same region. Figure 1(c) presents the region distribution, whose lifetime is less than 5 seconds, of the Docker workload. We can see that a huge number of regions are short-lived.

Observation # 2: Partial page hashing need to be adaptive

XDE [18] has demonstrated that partial page hashing can improve scan performance. We further observe that hashing a fixed number of bytes for each page, albeit partially, can limit the benefits of partial page hashing because different scenarios may have drastically different workloads.

For example, image display application renders a dotted image with the same color background. In this case, we need to hash more bytes in order to differentiate highly similar (but not identical) pages to avoid time-consuming byte-by-byte page comparison. While for other workloads, pages may be quite different to each other. An crypto application tends to hold memory regions with encrypted data as content. Hashing one or two bytes is already enough to identify the difference between pages.

Real world systems may be filled with all kinds of workloads. The workload might even evolve with time. For example, a container may hold a remote desktop, the user may close a paint application and open a https browser.

4 Overview

UKSM consists of two unique components, that is, memory region hierarchical distilling (in Section 5) and adaptive partial page hashing (in Section 6). Figure 2 demonstrates how these two components identify duplicated pages with minimal scanning overhead through a simple example. There are nine memory regions (R_0 - R_8). Figure 2(a) and 2(b) demonstrate two whole memory sampling rounds (*i.e.*, round 1 and 10).

Memory region hierarchical distilling manages memory regions by levels. There are *N* levels as shown in Figure 2, and every region falls into one of the *N* levels. Level *N* is the highest level and level 1 is the lowest level. A higher level has a higher scanning speed hence a higher sampling density. Let each gray bar represents a sampled page. Demonstrated in the figures, the sampling interval decreases as the level increases. Each newly allocated memory region is first inserted into level 1 of the hierarchy. Newly added pages may not be statically-deduplicated. Computing power should not be invested on these regions before they are proved worthwhile. That is why we put them into the lowest level of the hierarchy. During each sampling round, every memory region is sampled and filtered with a group of distilling parameters to decide whether it should be "promoted" to the next higher level or "demoted" to the next lower level . If all duplicated pages in a region are merged at some scan level, it goes back to level 1. If a region is unmapped it will be tagged and removed from the linked level later by the scanner. For example, in round 1 of Figure 2(a), regions R_3 and R_8 reside in level 2 and *N* respectively. When the scan thread proceeds to round 10, regions R_3 has been promoted to level *N* while R_8 has been demoted to level 2. Further elaboration of this technique are discussed in Section 5.

To further minimize the computational overhead, UKSM introduces the Adaptive Partial page Hashing (APH) approach. The key idea is that we will adjust a global hash strength after each global page sampling round in order to achieve a more cost-effective scanning. For example, in Figure 2, each star represents a hashing byte in each page. In round 1 of Figure 2(a), the hash strength is one byte per page. Based on the feedback from preceding sampling rounds, in round 10 of Figure 2(b), we increase the hash strength to two bytes per

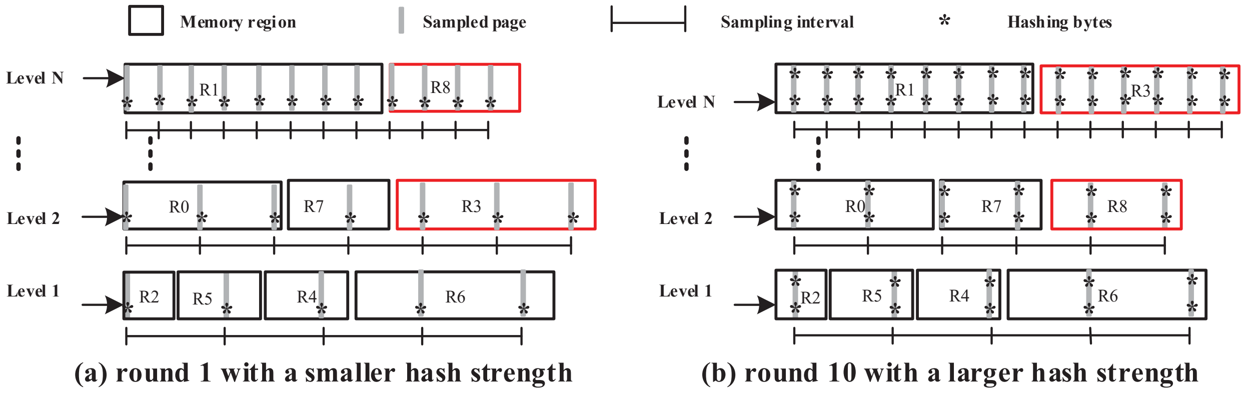
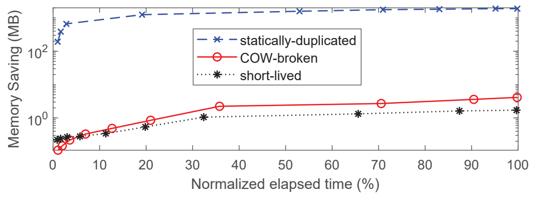
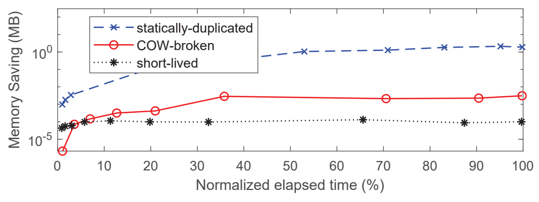


Figure 2: Memory region hierarchical distilling in two sampling rounds with different hash strength.



(a) Memory saving



(b) Mix with sparse background

Figure 3: Results with different workloads (Y-axis in each figure is exponential).

page. This design significantly improves per-page scan speed so that the overall system can respond to emerging duplicated pages rapidly and at the same time remains very low CPU consumption when no good candidates for deduplication exist. Details about hashing strength and feedback controls are discussed in Section 6.

5 Hierarchical Region Distilling

This section introduces design details of memory region hierarchical distilling. By analyzing the deduplication gain and lose of each kind of memory regions, we discuss distilling and scan principals (Section 5.1). Then whether a memory region is "promoted" or "demoted" is dominated by a set of threshold values (Section 5.2). At last, we present the hierarchical sampling procedure (Section 5.3).

5.1 Memory region characterization

Intuitively, a memory region should contain many statically-duplicated pages in order to be deduplication-friendly. In contrast, the unfriendly one could be frequently COW-broken, short-lived, or contain little identical content.

We uses three metrics to study the deduplication effects over a specific kind of regions. The first two metrics measure the gain and lose associated with deduplication, which are *memory saving* and *CPU consumption*, respectively. The third metric is *performance impact*, which reflects the slowdown ratio a deduplication method brought to a running application. Note this parameter is more comprehensive than the CPU consumption metric because a slowdown can result from cache/memory contention even if the deduplication worker (*e.g.*, *ksmd* – the kernel thread worker of KSM) executes on a dedicated CPU core. Ideally, we would like to maximize the first metric and minimize the other two.

This section utilizes four application configurations to emulate different workloads. Particularly, we use a memory footprint of about 2 GB for each of the *statically-duplicated*, *COW-broken* and *sparse* workloads. The compiling of the Linux kernel serves as an benchmark for *short-lived* workload which consumes about 30 MB memory space (only the anonymous pages, not including the file cache). We take KSM as the representative of existing CBPS approaches to demonstrate the complexity of balancing among three metrics. We record the maximum time needed for deduplicating all eligible pages of the *statically-duplicated* workload when using 100% capacity of a single CPU core. Then we normalize the time in x-axes of Figure 3, to demonstrate the progress of deduplicating each workload.

Memory saving vs. CPU consumption. Figure 3(a) shows how average memory saving progresses with time for *statically-duplicated*, *COW-broken* and *short-lived* workloads. Firstly, more CPU consumption does bring more memory saving, until the last duplicated page is merged. The deduplication speed, which is the slope of each line, drops rapidly as time goes on for *statically-duplicated* workload. Secondly, the memory saving of *statically-duplicated* workload is two orders of magnitude higher than that of *COW-broken* and *short-lived*

Table 1: Distilling and sampling parameters

V_{cow}	Only regions whose COW-broken ratios are lower than this threshold can be promoted.
V_{dup}	Only regions whose duplication ratios are larger than this threshold can be promoted.
V_{life}	Only regions lived longer than this threshold can be effectively scanned.
T_s	The sleep time in each sleep-scan cycle of the scan thread.
t_l	The expected time of sampling round for level l (in seconds).
t	The expected time of a global sampling round (in seconds).
p	The invested CPU percentage).
s	The estimated CPU cost of sampling one page.

workloads, which is consistent with our expectation. So we conclude that, for *statically-duplicated* workloads, invested computation is effective at the beginning. After all candidate pages are merged, further scan needs to be slowed down. For dynamic workloads (*COW-broken* and *short-lived*), higher CPU consumption is required to save the same amount of memory. Users may need to decide if the trade-off is worthwhile.

In Figure 3(b), we let each workload mixed with a *sparse* workload. The amount of memory saving decreases significantly. It is clear that scan of *sparse* regions in a system should be delayed, if not totally avoided, as much as possible.

Performance impact. We further study the performance impact to CPU intensive workloads brought by this hash-based KSM. One workload is a full SPEC-CPU2006 benchmark, and the other is the COW-broken Linux compiling workload mentioned above. If the scanning thread works at full speed with enough CPU resources (*i.e.*, scanning thread and workload threads each has its dedicated CPU core), the performance impact to *COW-broken* workload is 29.7%, and the impacts to other workloads range from 1.5% to 22.9%.

In-depth profiling shows that: 1) even with abundant CPU cores to separate workloads and the scanner, intensive scanning of CPU bound workloads makes the scanning thread contending more for memory management locking (*i.e.*, VMA locks, page table locks, etc), which introduces higher overhead for these workloads; 2) deduplication on frequently COW-broken pages may not bring much memory saving, but will bring many COW-broken page faults on merged pages, thus deteriorate performance.

5.2 Candidate region identification

The key characteristics (*i.e.*, *COW ratio*, *duplication ratio*, and *average page lifetime*) that indicate the duplication qualities of each memory region should be obtained first. This section introduces corresponding quantitative threshold values that can decide whether we "promote" or "demote" a memory region. Table 1 details these three thresholds, i.e., V_{dup}, V_{cow} and V_{life}. In particular, a regions with duplication ratio above V_{dup}, COW-broken ratio below V_{cow} and life longer than V_{life} can be identified as a good candidate for *statically-duplicated*. To control CPU overhead, we make the scanner work in a sleep-scan cyclic pattern with sleep time T_s. This parameter is related to the life time threshold.

For a memory region, the first parameter duplication ratio is estimated by dividing the duplicated page counter by the number of the pages sampled in this round. To compute its COW ratio, we need to obtain the number of COW-broken page faults on merged pages during each sampling round. This information can be easily obtained by hooking the page fault handler function. The last parameter lifetime is decided by the sleep time T_s and the sampling round time t (sum of t_l for each level in Table 1). Only those regions which live across this sleep-scan cycle time may get sampled.

How to choose threshold values? Threshold values of *duplication ratio, COW-broken threshold, lifetime* are critical parameters for UKSM. We design UKSM as a general system and it targets a wide range of scenarios. The default settings of *10%, 50%, 100ms* are obtained empirically and are shown to work well for a wide range of systems. The global sampling round time can be configured in the range of 2 - 20s with further details explained in Section 7. However, we also leave these parameters configurable for expert users who can tune UKSM to meet their application-specific needs. As far as we know, many follow-up production systems extend various configurations of UKSM to meet their particular needs [7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. The auto-tuning of these parameters could be a future work.

Why use the same set of threshold values across all levels? The higher the level is, the higher the page scanning frequency is. There is a larger chance of *false positive* in a lower level, where a low duplication ratio region may accidently get promoted. Hence, even with the same set of threshold values at all levels, we can successfully demote false positive regions and promote real positive regions. Further, UKSM performs merge together with the scanning. Every time a new duplicated page is found, besides being counted in the duplication ratio calculation, the page is merged directly. Statistically, even with the same threshold, regions in a higher level should have a larger duplication ratio than those in a lower level.

```
For(;;){
    A global sampling round {
        Sleep $T_s$;
        promote/demote regions;
        For(l = 1, l<= N, l++) {
            scan level l with budget time $t_l$;
        }
    }
}
```

Figure 4: Workflow of UKSM hierarchical sampling.

5.3 Hierarchical sampling procedure

Now that UKSM has the information of the key features of each memory region and the promotion criteria, this section discusses our hierarchical sampling approach which manages memory regions by levels and each region only belongs to one level. For instance, Figure 2(a) and 2(b) manage 9 regions by N levels. Figure 4 shows the workflow.

Scan a level When sample a specific level, all memory regions are grouped to be one flat linear space. The memory scanner starts at page offset of zero in this linear space and picks sample points by the length of *interval*. Note that a higher level possesses a smaller interval. If a sample point falls in a region, one page will be selected from this region. Particularly, we introduce a region specific offset permutation scheme to avoid sampling the same page repeatedly. For instance, although R_2 from level 1 is sampled in both rounds of Figure 2(a) and Figure 2(b), different pages are picked.

Once the page is selected, our scanner will get the page's hash value according to current hash strength (*i.e.*, bytes hashed in each page), and looks it up in two red-black trees trying to find a collision. One of the red-black trees (Tr_s) tracks the "merged" pages whereas the other one (Tr_{us}) records the "unmerged" ones. If the sampled page has an identical page in Tr_s, we increase the region's counter by one. If the sampled page is found to be identical to one of the pages in Tr_{us}, we move the page to Tr_s and increase the counters of both regions. Eventually, we update the page table and release redundant pages accordingly. UKSM keeps "merged" and "unmerged" page hashes in separate trees because merged pages should be managed in a read-only tree. Write to any node in this tree causes a COW operation.

This scan continues until the sample point reaches the boundary of the linear space. We call it a sampling round in this level. The scanner then proceeds to the next level.

A global sampling round A global round is finished after the level N sampling. then the scanner restarts from level 1. After each global round, the scanner estimates each region's duplication and COW-broken ratios. It is easy to see that with sufficient lifetime, every page of a memory region will be scanned. If a region is *unmapped* before every page is scanned, it will be removed from that level.

Sampling time control For each level, we can easily get the number of pages in one level as L_l. With invested CPU computation p and the estimated time of sampling one page s, we get the average page processing speed as p/s. Assuming the expected sampling round time for this level is t_l, the number of sample points in one round is $n = t \cdot p/s$. The sampling interval in each level is determined by $L_l/n = L_l \cdot s/(t_l \cdot p)$. The sleep time is T_s, so the active time of each sleep-active cycle is $T_s \cdot p/(1-p)$. Then we can get the number of pages to scan during each active cycle as $T_s \cdot p/(s \cdot (1-p))$.

In summary, users can configure two parameters which are p and t, as the invested CPU computation time and global sampling round time, respectively. According to our empirical study, p and t can be configured in the ranges of 0.2% - 95% and 2 - 20s, respectively.

6 Adaptive Partial Hashing

We propose a new page hashing function to reduce per-page scan and deduplication cost. The key idea is to *partially hash* a page. if the hash value is already sufficient to distinguish different pages, we do not need to hash a full page. Generally, the new hash function should have the following features:

- The hash strength (i.e. bytes hashed in a page) should be adjustable. If the memory pages are "quite different", a weaker strength is used. Otherwise, a stronger strength is applied.
- With the strongest strength, the hashing function should have a comparable speed and collision rate to SuperFastHash for arbitrary workloads.
- With weak strength values, the hash function should be significantly faster than SuperFastHash.
- The hash function should be bidirectional progressive with cost proportional to the delta of strength, hence the page hash values with an updated strength can be incrementally computed from previous values.

6.1 Hash strength adaptation

A weak hash strength may increase the possibility of false positive, which can result in additional overhead on *memcmp*. For each sampling round, we quantify the *profit* for using some hash strength by the time saved compared to that of using the strongest strength. We quantify its *penalty* by the additional time of *memcmp* due to collision. The calculation for both *profit* and *penalty* is instrumented in the scan functions. The aim of hash strength adaptation is to maximize the overall benefit of *profit-penalty*. In what follows, we explain how our adaptive algorithm finds the optimal strength for the hash function.

When the system starts up, the hash strength is initialized with half of the strongest strength. After the

```
#define STREN_FULL (4096/sizeof(u32))
u32 shiftr, shiftl;
u32 random_offsets[STREN_FULL];
u32 random_sample_hash(u32 hash_init,
    void *page_addr, u32 strength) {
    u32 hash = hash_init;
    u32 i, pos, loop = strength;
    u32 *key = (u32*)page_addr;

    if (strength > STREN_FULL)
        loop = STREN_FULL;
    for (i = 0; i < loop; i++) {
        pos = random_offsets[i];
        hash += key[pos];
        hash += (hash << shiftl);
        hash ^= (hash >> shiftr);
    }
    return hash;
}
```

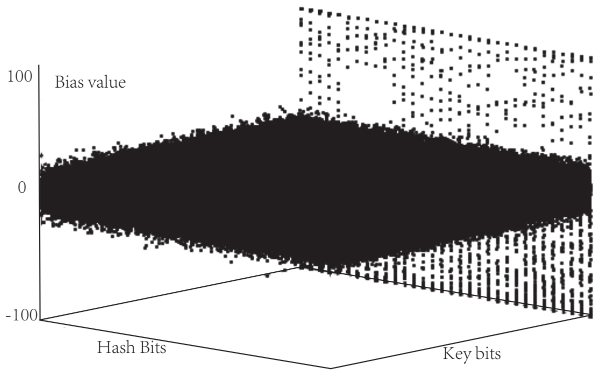
Figure 5: Progressive hash procedure.

first sampling round, the system enters a "probing" state trying to search for a strength that leads to a better overall benefit and finally stays in a dynamically "stable" state. The searching in the probing state simulates the TCP slow start process. The system firstly decreases the hash strength by a size variable named *delta* (initialized with 1) and checks if this change results in larger benefit. If it does, the system goes on trying until the benefit begins to decrease. During this process, *delta* will be doubled each time till the max value 32. Then the system records the maximum benefit point achieved and reset *delta* to 1. Similarly, the system will search in the other direction when increasing the hash strength. Once the system reaches an optimal point, it enters a stable state.

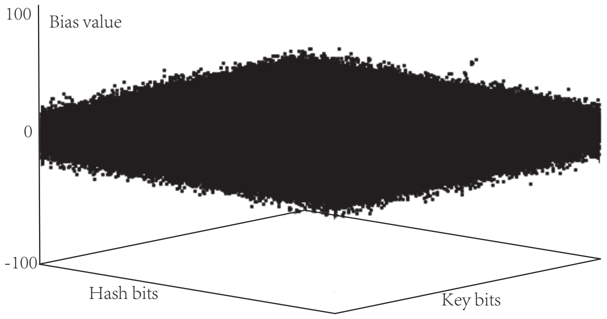
The state changes from *stable* to *probing* is triggered by one of the following conditions: 1) The benefit of the last sampling round deviates more than 50% from the benefit value when the system enters a stable state; 2) There is no *memcmp* caused by hash collision in the last two sampling rounds; 3) Every 1000 sampling rounds have been passed.

6.2 Progressive hash algorithm

We decide to use random sampling to fulfill the feature of dynamically adjustable strength. A universal random permutation of all the 32-bit-aligned offsets in a page is computed when the deduplication system is initialized. This is important because randomization is necessary in cases where some pages have specific patterns (*e.g.*, leading zeros). When a page is hashed with strength *I*, only the first *I* 32-bit data units are read and calculated from the page with the corresponding offsets in the permutation. In order to limit the execution time for the strongest strength, we derive the hash algorithm based on Jenkins's "one-at-a-time hash" [35] which is also the ancestor of SuperFastHash. The algorithm framework is shown in Figure 5. In the code, *random_offsets* is the buffer holding the random permutation of offsets; *shiftl* and *shiftr* are the two values we need to parameterize to further satisfy other features required for collision rate and incremental/decremental calculation; *STREN_FULL* is the strength for hashing a full 4 KB page content.



(a) SuperFastHash avalanche



(b) Random_sample_hash avalanche

Figure 6: The avalanche effects over a 4KB page.

Achieve low collision To ensure a low collision rate, we study the avalanche effect [36, 37, 38] of the hash function in our algorithm when hashing a full page with different *shiftl* and *shiftr* values. Avalanche is a desirable property of hash algorithms to achieve low collision rate wherein if the input is changed slightly the output can change significantly in pseudo-random manner. We evaluate the avalanche effect with an initially zeroed two dimensional matrix which we call avalanche *bias_matrix*. Given a randomly generated key of page size, we flip the *i*-th bit. If this operation leads to the flipping of the *j*-th bit of the hash value, we increase point (i, j) in *bias_matrix* by one, and if the *j*-th bit of hash value is not affected, we decrease *bias_matrix(i, j)* by one. This process is repeated for multiple times, then we calculate the average value of $bias\_matrix(i, j)$ for all $i \in [0, 32767], j \in [0, 31]$. Ideally, one bit changes in the key will affect the output of hash value with 50% probability. Therefore, the corresponding *bias_matrix* entry should approximate 0 on average.

Figure 6(a) is the 3D visualization for such an avalanche *bias_matrix* of SuperFastHash. We can see that most of the points are closed to the $bias\_value = 0$

```
u32 reverse_addeq_shiftl(u32 n) {
    u32 ret = n, turn = 1;
    n <<= shiftl;
    while (n != 0) {
        if (turn)
                ret -= n;
        else
                ret += n;
        turn = !turn;
        n <<= shiftl;
    }
    return ret;
}

u32 reverse_xoreq_shiftr(u32 n) {
    u32 ret = n;
    n >>= shiftr;
    while (n != 0) {
        ret ^= n;
        n >>= shiftr;
    }
    return ret;
}
```

Figure 7: Reverse functions for progressive hash.

plane except for the last several key bytes. We therefore evaluate the avalanche effect of the hash algorithm by the number of the "bad points" whose deviation from the $bias\_value = 0$ plane exceeds a threshold (for our case, we take 50). We conduct an exhaustive search of all possible (*shiftl*, *shiftr*) value pairs and generate a priority list of them (omitted due to space limitation). The avalanche behavior of our hash algorithm with maximum strength is illustrated in Figure 6(b). It is better than that of SuperFastHash as illustrated.

Achieve progressive hashing Assume the recorded hash value of a page is achieved at strength S_1 but the current strength is S_2, the updated hash value can be achieved with additional computation using the recorded result for S_1. If $S_2 > S_1$, this can be done by filling the *hash_init* parameter (in Figure 5) with the hash value at strength S_1. If $S_2 < S_1$, the hash calculation must be reversed. The "+=" operation can be reversed with "-=". The "+=" and "^=" operations combined with "<<" and ">>" can be reversed by the code in Figure 7.

Small values of *shiftl* and *shiftr* will increase the cost of reverse operations. We choose the pair of (19, 16) from the priority list for (*shiftl*, *shiftr*) which brings very good avalanche effect and at the same time makes the cost of the reverse operation comparable to that of progressive hash operation. We compare the speed of *random_sample_hash* with maximum strength and SuperFastHash and find that our algorithm is only about 2% slower than SuperFastHash. The final avalanche effect result of our hash algorithm with maximum strength is slightly better (fewer "bad points" as we state above) than that of SuperFastHash.

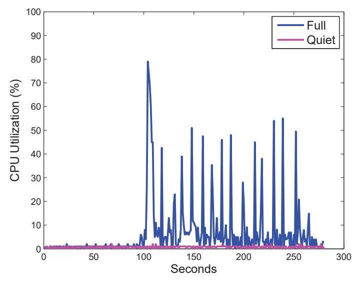
Why use a global hash strength design instead per-region or per-app hash strength? Here hash strength denotes how many 32-bit words are hashed to generate a fixed length hash value. If we use different numbers of 32-bit words for two different pages, these two pages cannot be compared directly. That is why we use a global hash strength, so that every pair of pages can compare their hash values directly.

Why develop APH based on SuperFastHash? There are some newly developed fast hash algorithms, such as Spooky [39], xxHash [40], and Murmur [41], which are much faster than SuperFastHash. Whether an algorithm can derive an adaptive version depends on its design details. Using one of those hashes in our adaptive hash framework could be an interesting future work.

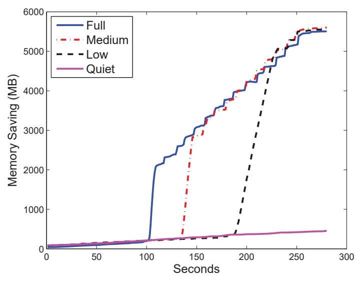
7 Implementation and Configuration

UKSM is implemented in both Linux kernel and Xen, each with more than 6,000 lines of C code. In Linux, UKSM hooks the Linux kernel memory management subsystem for monitoring the creation and termination of application memory regions. The kernel page fault routine is also hooked to log COW-broken events in each region. UKSM scanner is created as a kernel thread *uksmd*. In Xen, UKSM scanner is implemented as a softirq service routine of the Xen hypervisor. The Xen memory management subsystem is also hooked in the same way as in Linux kernel.

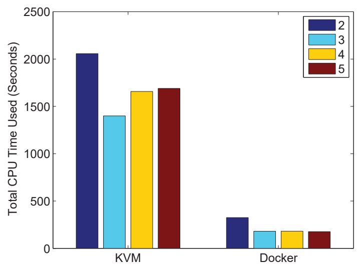
To facilitate drop-in utilization of UKSM, we borrow the idea of "CPU governors" with which the Linux kernel simplifies the configuration for Intel CPU frequency [42]. We define several default parameter sets named as "governors" to represent "how aggressive" the scanner should be. These "governors" are *Full*, *Medium*, *Low*, and *Quiet*. With the *Full* governor, it can use up to 95% CPU and finishes one global sampling round in 2 seconds. From *Full* to *Low*, each governor doubles the global sampling round time and reduces the top CPU usage by half. The *Quiet* governor is designed to be used in battery powered systems where workloads are static most of the time (*e.g.*, Android). It has a top CPU consumption of 1% and a global sampling round time up to 20 seconds. We use the workload of *booting 25 VMs* in Section 8 to depict the performance metrics of UKSM under different governors. The results are shown in Figure 8(a) (plotting only *Full* and *Quiet* for clarity) and Figure 8(b). We can see that with the *Low* governor, UKSM can already catch up with the booting process (about 260 seconds). The main difference is how fast a governor can catch up. We also observe that the CPU cycles consumed by the governors are proportional to the number of pages they deduplicate. It is consistent with our design purpose. Since the *Full* governor is more responsive, we choose *Full* as the default governor and use it for all later evaluations unless specified otherwise.



(a) CPU consumption for governors



(b) Memory saving for governors



(c) Performance with different level numbers

Figure 8: *Performance metrics under different configurations*

For the scan levels, the bottom level serves as a baseline sampling with CPU consumption as low as possible. We believe 0.2% should be acceptable for general systems. The CPU consumption of the top level is given by the "governor". The CPU consumption for each intermediate level is halved. The sampling round time for each level is evenly derived from global sampling round time. The number of scan levels determines how smooth the promoting process could be: more levels will make a memory region more carefully sampled before it is intensively scanned. On the other hand, less levels will make the system response more quickly to emerging duplication but may suffer false positives caused by sampling singularity (*i.e.*, a region is falsely identified as "good" after one scanning round). We tested the configuration from 2 levels to 5 levels with real world benchmarks used in Section 8. As shown in Figure 8(c), for larger regions of the KVM workload, sampling singularities are less likely to happen. So 3-level-sampling is the best choice. For smaller regions of the docker workload, 5 level is the best choice. We choose 4 levels as UKSM default.

Till the time of this paper being written, the feedbacks from different sources have demonstrated that while the system design stems from a server environment, its design and parameters are shown to work in a wide range of environments such as a mobile system (*e.g.* Android). Only very few people adjusted the individual parameters according to their specific requirement. We leave comprehensive parameters tuning under different types of workload as our future work.

8 Evaluation

We evaluate our UKSM implementation in comparison with the Linux kernel KSM. The operating system for our benchmarks is CentOS 7 with vanilla Linux kernel 4.4. The hardware setting is Intel(R) Core(TM) i7 CPU 920 with four 2.67GHz cores and with 12 GB RAM. The benchmarks include emulated workloads and real-world workloads. For fair comparison, the native Linux KSM scanner is upgraded to use SuperFastHash, which has a better performance. Our evaluation centers around five key questions:

How efficient is UKSM on different workloads? Using emulated workloads each focusing on a single type, we show that UKSM can be up to 12.6× more efficient than KSM on densely/sparsely 1:1 mixed workloads and can be up to 5× more efficient than KSM on frequently COW-broken workloads (Section 8.1).

How flexible is UKSM with customization? On the same set of workloads, we show that UKSM can filter different types of memory regions with different thresholds. With UKSM, users can customize their trade-offs, while previous approaches like KSM cannot (Sections 8.1.2 and 8.1.3).

What is the performance v.s. overhead tradeoff of UKSM on production workloads? By experiments on KVM VMs and Docker containers, we show that UKSM significantly outperforms *ksmtuned*-enhanced KSM in VM benchmarks. It can deduplicate the typical setup of Docker containers (which cannot be handled by KSM) with negligible CPU consumption (less than 1% of one core). The results also prove that our approach outperforms XLH even without I/O hints. The experiments on desktop servers with mixed workloads shows that UKSM can deduplicate newly generated pages in seconds (Section 8.2).

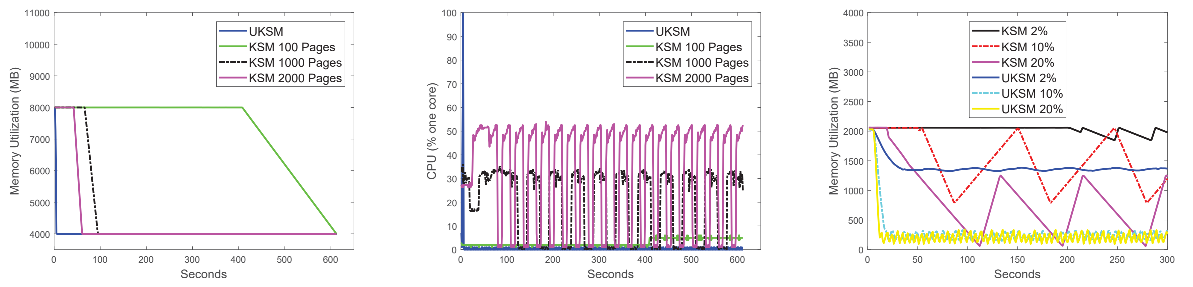
How does Adaptive Partial Hashing perform compared to non-adaptive algorithms? We analyze the effectiveness of APH on densely and sparsely duplicated pages. We find that APH alone can make the scanning speed of UKSM up to 7× that of KSM on typical cloud workloads (Section 8.3).

How large is the application penalty of UKSM? For native environments, UKSM's penalty is less than 3%. For virtualized environments, UKSM's penalty is less than 1.8% (Section 8.4).

8.1 Deduplication efficiency analysis

8.1.1 Statically mixed workload

We first evaluate the deduplication efficiency of UKSM and KSM on a static workload. This workload is composed of two programs. Each program creates 4 GB memory. One fills memory with identical page data. The other program fills memory with random data. After they complete filling pages, we start the UKSM/KSM



(a) Deduplication speed and memory saving (b) CPU consumption (c) Memory Utilization for COW-broken

Figure 9: *Benchmark performance comparison.*

daemon. Since the default scanning speed (100 pages each cycle) of KSM is very low, we also obtain the results of KSM when scanning 1000 and 2000 pages each cycle. As illustrated in Figure 9(a), it takes only 5 seconds for UKSM to merge all duplicated pages. While the deduplication time for KSM at 100, 1000, 2000 pages are 611, 95, 61 seconds respectively.

We then analyze the CPU usage of UKSM and KSM in the above benchmark. As shown in Figure 9(b), the CPU consumption pattern of UKSM is composed of very thin spikes and with average CPU of less than 1%. UKSM only reaches its peak CPU consumption (around 95%) at the 5th second. KSM constantly demonstrates very high CPU consumption especially at high scanning speed. This phenomenon reflects the fact that UKSM reacts rapidly to emerging duplicated pages and has a very low background CPU usage (recall that the pre-defined value for sampling level 1 is 0.2%) when all duplicated pages are already merged.

We then calculate the deduplication efficiency as *memory_saving* over *deduplication_CPU_consumption*, where *deduplication_CPU_consumption* is the sum of CPU consumption ratios of each second before all pages are deduplicated. From calculation, we find that UKSM is 8.3×, 12.6×, 11.5× more efficient than that of KSM at scan speed of 100, 1000, 2000 pages respectively.

8.1.2 COW-broken workload

We then demonstrate how UKSM improves over KSM on frequently COW-broken workloads. We emulate this case with a program that maps 2GB of memory and repeatedly *memset* one full page (with the same content) every 10 ms from the start to the end of the region. With the default setting of UKSM (COW-broken ratio threshold of 50%), it totally avoids intensively scanning this workload. However, UKSM can be configured to scan this workload if we disable the COW-broken filtering (note that KSM cannot be customized to avoid scanning this workload).

We make both KSM and UKSM consume about the same CPU power (2%, 10% and 20% of one core) and then compare the memory saving of them as shown in Figure 9(c). We can see that KSM saves only about 1/3 to 1/5 of the memory that could be saved by UKSM. Furthermore, the performance of UKSM is quite stable, in contrast, the memory saving of KSM suffers from large variations.

8.1.3 Short-lived workload

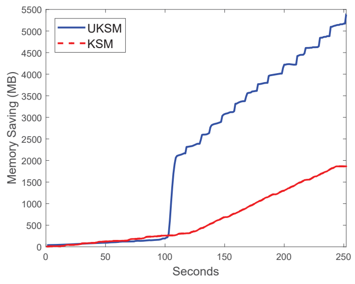
We emulate this case by a program that infinitely repeats a cycle of "mmap a region of 500MB pages of the same content, sleep for time of T, then unmap this region and sleep for another T". We observe that even with very aggressive settings of KSM (sleep time sets to 20ms, *pages_to_scan* sets to 2000, consuming about 50% CPU), it cannot merge a single page if T is less than 2 seconds.

Although it totally filters out this case with its default settings, it is possible to make UKSM sensitive to short-lived pages. After we assign its sleep time to 20ms, its max CPU consumption to 50% and its sampling round time of each level to be 50ms, 20ms, 10ms, and 5ms, respectively, UKSM can merge almost all the pages even if T is less than 200ms.

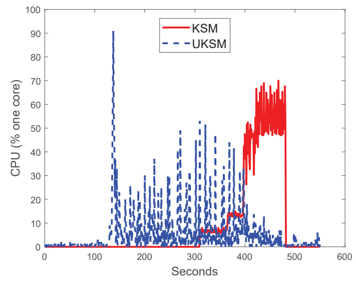
8.2 Real world benchmarks

8.2.1 KVM virtual machines

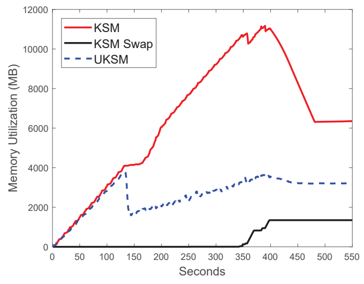
Booting 25 VMs with abundant memory We uss the same benchmark used in XLH [27], As XLH is not an open-source implementation, we use an almost identical hardware/software platform settings. Thus we can compare our results with theirs. We booted 25 VMs (installed with Ubuntu server 16.04) each with a single VCPU and 512MB of memory in parallel, with starting time of 10 seconds apart. KSM is configured with the settings as that in XLH. UKSM uses the default settings. After about 260 seconds, all VMs are fully booted. Up to this point, UKSM has merged 5.3GB of memory, about 3× of what KSM has merged (Figure 10(a)). We need a warming up time to build rb-tree, offset and figure out duplications. That explains why in around 100 sec we have a jump. KSM and UKSM use about the same amount of CPU resources during this process. [27] reported that XLH can achieve only 2× the memory saving compared to KSM with same CPU resources. This implies that UKSM outperforms XLH significantly.



(a) 25 VMs Memory Saving

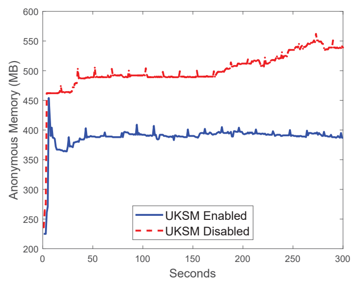


(b) 40 VMs CPU

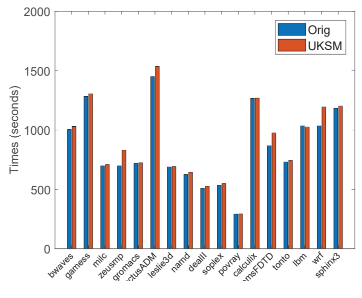


(c) 40 VMs Memory Utilization

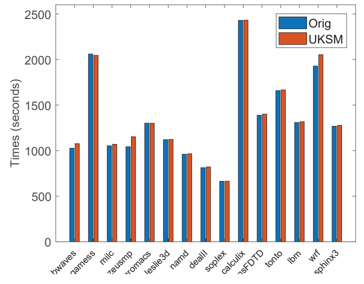
Figure 10: *Real workload performance comparison.*



(a) Deduplication for Docker containers.



(b) CPU2006 fp, 1 CPU core



(c) CPU2006 fp benchmarks in KVM

Figure 11: *(a) Deduplication for Docker containers; (b), (c) Performance impact of UKSM.*

Booting 40 VMs with memory overcommit We boot 40 VMs (with a single VCPU and 1GB of memory) in parallel, with starting time of 10 seconds apart. We record the total boot time, memory and swap usage until the system stabilized. In this benchmark, we compare UKSM (with default settings) with KSM settings adjusted by *ksmtuned*. As mentioned in related work, *ksmtuned* can dynamically increase and decrease the KSM scanning speed according to memory usage. Figure 10(b) illustrates the CPU utilization of UKSM and KSM during the deduplication process; from calculation, the aggregated CPU utilization of UKSM is about half of KSM. Figure 10(c) shows that, with KSM and *ksmtuned*, the system triggers about 1.3GB of swap and this slows down the boot process of the VMs significantly (from about 420 seconds to 550 seconds). UKSM uses only half of physical memory and requires no swap usage when the system stabilizes. The peak memory usage with UKSM is only 32% of that with KSM. With KSM, the last VM booted still waits for a long time before it can be logged in after *ksmtuned* shuts down KSM at 480 seconds. Moreover, when the swapping storm is triggered, the system suffers bad responsiveness, which would be a devastating user experience.

8.2.2 Docker containers

We start 3 Docker containers each running a WordPress website in LAMP environment (*i.e.*, Linux, MySQL, Apache and PHP). Each website contains a page with the same set of images and texts. Then we uses Firefox to emulate normal user connections by making it refreshing each website's page every second. Figure 11(a) shows the amount of the anonymously mapped memory in this process when UKSM is enabled or disabled. We find that the average memory used with UKSM enabled is only 61% of that when it is disabled. The CPU consumption during this process of *uksmd* is mostly less than 1% (one core) with few spikes to around 3%. At the time of this paper is written, no other open source implementation to our knowledge can deduplicate memory of containers, hence we do not compare with others in this benchmark.

It's worthy to note that Docker containers try hard to share the underlying files with aufs [43]. There may not be that much duplication in file cached pages. UKSM only handles anonymous pages for containers by now. File cache deduplication is left as our future work.

Table 2: UKSM introduced space saving and time consumption for mixed workload.

Application	Okular PDF	Firefox	FlashPlayer	GIMP
Space saving (MB)	415	27	63	39
Time (s)	2	5	4	1

8.2.3 Mixed workload on desktop server

In this Ubuntu desktop server, we run four applications, *i.e.*, Okular PDF reader, Firefox browser, FlashPlayer, and GIMP painter, simultaneously. The default memory is around 1,248 MB. We then perform operations with each software separately. At the same, we record the time and deduplication gain of UKSM, by monitoring the *htop* tool. With UKSM, it takes only 2/5/4/1 seconds to deduplicate 415/27/63/39 MB memory for

Okular/Firefox/FlashPlayer/GIMP in this mixed wowrkload environment. Consistent with our design principal, UKSM works well in real world systems.

8.3 Analysis of Adaptive Partial Hashing

In the first two experiments, we use two extreme scenarios, one system contains no duplication, and one full of duplicated pages. Thus, UKSM hierarchical region distilling has no effect at all, since all regions are at either the highest level, or the lowest level. In this case, the only difference between UKSM and KSM is that UKSM turns on APH.

8.3.1 Effectiveness of Adaptive Partial Hashing

Scanning speed in regions with low redundancy We first demonstrate how fast the optimized system can scan regions with low page redundancy. One extreme case is when the system is full of "quite different" pages, that is, no two pages have the identical 32-bit words at the same offset. That makes the hash strength drop to 1, or in other words, the hashing cost is about 1/1000 of the SuperFastHash hashing. The maximum scanning speed of UKSM in this case is about 5.9× higher than that of the hash-based KSM or 7.4× higher than the original KSM. On the other hand, if the pages are quite similar but not equal, the strength of hash function may rise to 1,024 words. In this worst case, the maximum scanning speed is about the same as the hash-based KSM, which is expected by the fact that the hash algorithm with the strongest strength is comparable to SuperFastHash.

Deduplication speed on highly redundant regions We then measure the maximum speed for merging identical pages when hash strength is 1. It's approximately 2.5× higher than that of the hash-based KSM or 7× higher than that of the original KSM. When increasing the strength of hash function to the maximum value, the merge speed becomes comparable to hash-based KSM or about 3× that of the original KSM.

8.3.2 Strength of hash function

The actual scan/deduplication speed on real workloads depends on the memory content pattern and our system adapts its page hash strength accordingly. A comprehensive testing on a variety of workloads has shown that the hash strength is usually within 100 words (recall that the highest value is 1,024). The scan speed of UKSM at this hash strength is about 6 to 7× higher than that of the original KSM or 2 to 5× higher than that of the hash-based KSM. We expect even smaller strength values in scientific computing or data processing environments. For example, the hash strength for all 12 SPEC-CPU2006 benchmarks ranges from 2 to 17, with the average of 7.5. These results validate the effectiveness of our hashing design.

8.4 Performance Impact

We evaluate the runtime overhead of UKSM using CPU intensive workloads of Standard Performance Evaluation Corporation (SPEC) CPU2006 benchmarks with *uksmd* under the *Full* governor. The experiments are firstly run on the host and then inside KVM virtual machines.

CPU2006 on host We evaluate UKSM in two scenarios, where 1) UKSM and CPU2006 are running within one CPU core; 2) The system has enough CPU resources so that UKSM will not compete with CPU2006. The result of CPU2006 float point benchmarks in the first scenario is shown in Figure 11(b). We can see that the average overhead is less than 3.0%. The average overhead in the second scenario is about 1.5%. We observe similar results with CPU2006 integer benchmarks (2.7%). We do not show the other figures due to space limitation.

CPU2006 in KVM The benchmarks run inside 2 VMs. Two CPU cores are enabled on the host for VMs. Each VM is assigned with 1 VCPU. As illustrated in Figure 11(c), the average overhead for CPU2006 float point is 1.8% (0.9% on CPU2006 integer group).

It is worth noting that these results are the worst case upper bound under the *Full* governor. We achieved almost the same memory saving for these benchmarks under the *Low* governor with negligible overhead.

9 Conclusion

We design a novel memory deduplication system called UKSM that (1) samples the whole memory with an enhanced hashing scheme to estimate the duplication ratio and dynamics of each memory region; and (2) performs different scanning policies for different regions to maximize computation efficiency and minimize application penalty. Experiments on both emulated workloads and real-world benchmarks show substantial improvements compared to standard Linux KSM and I/O hints based approach XLH.

Acknowledgment

The authors would like to thank our shepherd Prof. Hong Jiang and anonymous reviewers for their valuable comments. This work was supported in part by the National Science and Technology Major Project of China under Grant Number 2017ZX03001013-003, the Fundamental Research Funds for the Central Universities under Grant Number 0202-14380037, the National Natural Science Foundation of China under Grant Numbers 61772265, 61370028, 61602194, and 61321491, the National Science Foundation under Grant Numbers 1547428 and 1738965, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

[1] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.

[2] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 244–249, 2011.

[3] Shashank Rachamalla, Debahuti Mishra, and Parag Kulkarni. Share-o-meter: An empirical analysis of ksm based memory sharing in virtualized systems. In *IEEE International Conference on High Performance Computing (HiPC)*, 2013.

[4] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. Determining the use of interdomain shareable pages using kernel introspection. *Department of Computer Science, Aalborg University*, 2007.

[5] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[6] Uksm official site. `http://kerneldedup.org/en/projects/uksm/`.

[7] Xanmod kernel 4.4.0 for ubuntu linux. `https://ubuntuforums.org/showthread.php?t=2307617`.

[8] Openmandriva lx 2014.2 kernel. `https://wiki.openmandriva.org/en/2014.2/New`.

[9] Achlinux port of linux-pf-lts 3.14.72-1. `https://aur.archlinux.org/packages/linux-pf-lts/`.

[10] Opensuse linux port of kernel-postfactum. `https://software.opensuse.org/package/kernel-postfactum`.

[11] Calculate linux 14.16. `http://distrowatch.com/?newsid=08899`.

[12] Shinto kernel secondreality v40a05. `https://www.precog.me/2015/02/27/release-shinto-kernel-secondreality-v40a05/3/`.

[13] Xda-developers, charm-kiss kernel 20140107. `http://forum.xda-developers.com/showthread.php?t=2487113`.

[14] Xda-developers, wonderchild kernel. `http://forum.xda-developers.com/showthread.php?t=2565299`.

[15] Xda-developers, decimalman's kernel playground. `http://forum.xda-developers.com/showthread.php?t=2226889`.

[16] Xda-developers, renderbroken's custom kernel. `http://forum.xda-developers.com/showthread.php?t=2724016`.

[17] Rodrigo Ceron, Rafael Folco, Breno Leitao, and Humberto Tsubamoto. Power systems memory deduplication. *IBM Redbooks*, 2012.

[18] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010.

[19] Paul Hsieh. The superfasthash function. `http://www.azillionmonkeys.com/qed/hash.html`.

[20] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Linux symposium*, pages 19–28, 2009.

[21] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Linux symposium*, volume 1, pages 225–230, 2007.

[22] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 15–26. ACM, 2012.

[23] Red hat enterprise linux virtualization administration guidechapter ksm. `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Administration_Guide/chap-KSM.html`.

[24] Anshuj Garg, Debadatta Mishra, and Purushottam Kulkarni. Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments. In *the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 44–59. ACM, 2017.

[25] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John CS Lui. Smartmd: A high performance deduplication engine with mixed pages. In *USENIX Annual Technical Conference (ATC)*, pages 733–744, 2017.

[26] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Ksm++: Using i/o-based hints to make memory-deduplication scanners more efficient. In *ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE)*, 2012.

[27] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Xlh: More effective memory deduplication scanners through cross-layer hints. In *USENIX Annual Technical Conference (ATC)*, pages 279–290, 2013.

[28] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. In *ACM SIGPLAN Notices*, volume 48, pages 51–62, 2013.

[29] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. Cmd: classification-based memory deduplication through page access characteristics. In *ACM SIGPLAN Notices*, volume 49, pages 65–76, 2014.

[30] Yinjin Fu, Hong Jiang, Nong Xiao, Lei Tian, and Fang Liu. Aa-dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 112–120. IEEE, 2011.

[31] Feng Chen, Tian Luo, and Xiaodong Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *USENIX File and Storage Technologies (FAST)*, volume 11, pages 77–90, 2011.

[32] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–9. IEEE, 2009.

[33] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *USENIX Annual Technical Conference (ATC)*, volume 2012, pages 285–296, 2012.

[34] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *USENIX File and Storage Technologies (FAST)*, volume 9, pages 111–123, 2009.

[35] Bob Jenkins. A hash function for hash table lookup. `http://www.burtleburtle.net/bob/hash/doobs.html`.

[36] Wikipedia - avalanche effect. `https://en.wikipedia.org/wiki/Avalanche_effect`.

[37] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228:15–23, 1973.

[38] Edward Dawson, Helen Gustafson, and Anthony N Pettitt. Strict key avalanche criterion. *Australasian Journal of Combinatorics*, 6:147–153, 1992.

[39] B Jenkins. Spookyhash: a 128-bit non-cryptographic hash (2010). *http://burtleburtle.net/bob/hash/spooky. html*, 2014.

[40] Yann Collet. xxhash-extremeley fast hash algorithm, 2016.

[41] Austin Appleby. Murmurhash 2.0, 2008.

[42] Linux kernel intel p-state driver. `https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt`.

[43] aufs another unionfs. `http://aufs.sourceforge.net/aufs.html`.