nuKSM: NUMA-aware Memory De-duplication for Multi-socket Servers

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology (Research)
IN
Faculty of Engineering

BY

Akash Panda

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

August, 2021

Declaration of Originality

I, **Akash Panda**, with SR No. **04-04-00-10-22-18-1-16142** hereby declare that the material presented in the thesis titled

nuKSM: NUMA-aware Memory De-duplicatiton for Multi-socket Servers

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2018-21**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

DEDICATED TO

My Parents

and everyone who has made the work possible.

Acknowledgements

I want to thank all the people who contributed in some way to the work described in the thesis. First and foremost, I would like to acknowledge my indebtedness and render my warmest thanks to my research advisor, Dr. Arkaprava Basu, for accepting me into his group. During my tenure, his friendly guidance, expert and professional advice made the work possible. It has been invaluable throughout all stages of the work. I would also express my gratitude to Ashish Panwar for extended discussions and valuable suggestions that have contributed significantly to the work's improvement.

The thesis has been written during my tenure at the Computer Systems Lab (CSL) at the Department of Computer Science and Automation, Indian Institute of Science (IISc). I want to thank the Indian Institute of Science for providing excellent working conditions. I would want to thank my fellow lab-mates from CSL for making the lab a lively and fun place to work. I would also like to thank all the office staff in the Department of Computer Science and Automation (CSA). Their effort to make administrative tasks smooth and easy for all the members of the department is invaluable.

I would like to specially thank all those associated with IISc Cricket Club for making my life wonderful during my stay at IISc, which helped me concentrate on my work better. It was the place that provided me with the environment to forget a bad day at work.

Finally, I would like to thank my parents and my younger brother for their constant love, patience, and support throughout my journey. It would have been impossible for me to advance in work without their love and encouragement.

Abstract

An operating system's memory management has multiple goals, *e.g.,* reducing memory access latencies, reducing memory footprint. These goals can conflict with each other when independent subsystems optimize them in silos.

In this work, we report one such conflict that appears between memory de-duplication and NUMA management. Linux's memory de-duplication subsystem, namely KSM, is NUMA unaware. Consequently, while de-duplicating pages across NUMA nodes, it can place de-duplicated pages in a manner that can lead to significant performance variations, unfairness, and subvert process priority.

We introduce NUMA-aware KSM, a.k.a., nuKSM, that makes judicious decisions about the placement of de-duplicated pages to reduce the impact of NUMA and unfairness in execution. nuKSM also enables users to avoid priority subversion. Finally, independent of the NUMA effect, we observed that KSM fails to scale well to large memory systems due to its centralized design. We thus extended nuKSM to adopt a de-centralized design to scale to larger memory sizes.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Memory management is one of the most critical pieces in an operating system's design. It has several responsibilities ranging from ensuring quick access to data by applications to enabling memory consolidation. For example, judicious placement of pages in multi-socket NUMA (non-uniform memory access) servers could determine the access latencies experienced by an application [1, 4, 13, 38]. Similarly, memory de-duplication can play a pivotal role in memory consolidation and over-commitment [2, 9, 22, 39, 40].

Different responsibilities of memory management can conflict with each other [12, 23, 28]. This often happens when different subsystems of an OS are responsible for different memory management goals, and each works in its silo [12]. In this work, we discover how Linux's de-duplication efforts can conflict with its NUMA management goals.

De-duplication plays an important role in memory consolidation and over-commitment, particularly under virtualization [9, 22, 35]. It is not uncommon for different virtual machines (VMs) to run the same or similar OSes, libraries, and applications [8]. Consequently, many physical pages belonging to different VMs could contain the same content. When such instances of VMs run on the same machine, it provides ample opportunities to consolidate the aggregate memory usage through de-duplication of duplicate contents. Linux's Kernel Same page Merging (KSM [14]), VMware's Transparent Page Sharing (TPS [39]) are real-world examples of memory de-duplication subsystems.

Linux's KSM periodically scans contents of pages mapped in different process's [1] virtual address spaces, including those mapped by virtual machine's (here, KVM) guest physical address space. KSM checks if the content matches with other pages that it had scanned in the past. When two pages contain the same content, they are de-duplicated wherein one of them is

[1] We use VMs and processes interchangeably since VMs are KVM processes to Linux (hypervisor).

retained and the other is freed. All mappings to the retained page are rendered Copy-on-Write in all processes that share it to prevent one process from modifying another's data.

To enable a greater opportunity for de-duplication, KSM allows de-duplicaiton of contents across NUMA nodes. In NUMA servers, a portion of physical memory is *local* to a processor (*i.e.,* attached to the same socket) and thus, can be accessed quickly. The rest of the memory is attached to processor(s) on a different socket(s), *i.e.,* remote. A remote memory access could be 1.5–2× slower than a local access [1, 13]. For example, on our test platform with Intel Xeon Gold 6140 processors, we find the local access latency is 89 ns and remote access latency is 139 ns. Thus, for better performance, the majority of memory accesses should be local for a process.

De-duplication of memory across NUMA nodes inevitably induces some remote memory accesses in one or more VMs. When two pages on different NUMA nodes with identical contents are de-duplicated, KSM retains one of them and frees the other. The virtual addresses that were earlier mapped to the freed physical page would now be mapped onto the de-duplicated copy residing on another NUMA node. Consequently, subsequent accesses to the address range by the VM whose copy is freed would become remote.

Unfortunately, we find that KSM is unaware of the NUMA implications in modern multi-socket servers. While remote accesses are not completely avoidable for pages de-duplicated across nodes, NUMA-unawareness leads to unintended and uncontrolled performance variations and *unfairness* in execution. We demonstrate that one VM could experience significantly more remote accesses (*e.g.,* more than 90%) than another, even when both execute instances of the same application. Consequently, the performance (normalized runtime) of two identical virtual machines can differ by as much as 46%.

Upon de-duplication of pages across nodes, a VM's performance hinges on whether its page with duplicate contents is retained or is freed. KSM scans virtual address spaces of one process (VM) at a time to identify candidates for de-duplication (*i.e.,* mergeable). However, when it finds mergeable pages, the NUMA-ness is not considered in deciding which copy to retain. *The relative order in which KSM happens to scan virtual address spaces of the processes/VMs determines which node will host the de-duplicated pages.* The order of scan itself is dependent on the relative process/VM creation order and, thus, arbitrary.

We also find that KSM is unaware of the priority of different processes. When coupled with NUMA-unawareness, this leads to priority subversion whereby a higher priority process slows down more compared to a lower priority one. Importantly, users have no control over which processes or virtual machines should enjoy more local accesses to de-duplicated pages. Consequently, there is no way for Linux users to control the performance implications of NUMA while using system-wide de-duplication for better memory consolidation and over-commitment.

To this end, we introduce nuKSM– NUMA-aware KSM. The foremost objective of nuKSM is to make an *informed* choice on which of the pages with duplicate contents to retain and which one to free. Specifically, it strives to reduce the total number of remote accesses by keeping a de-duplicated page close to the VM that is accessing it more frequently. This is because the overhead of NUMA is incurred only when a process performs remote access. nuKSM leverages information that is already available about page accesses in the page reclamation subsystem of Linux (*e.g., active* and *inactive* lists) for this purpose.

It may be possible, however, that both VMs frequently access the pages being de-duplicated. In that case, nuKSM strives to spread the overhead of remote accesses equitably to the VMs. Specifically, in such cases, nuKSM distributes de-duplicated pages across nodes in a round-robin fashion. This significantly improves fairness and avoids performance variations. We find that nuKSM reduces performance variations in certain workloads from 46% to a mere 4%.

Finally, to avoid priority subversion, nuKSM enables a user to request enforcement of process priorities in NUMA overheads incurred due to de-duplication. Specifically, when enabled (through a flag), nuKSM ensures that the location of de-duplicated pages reflects the relative priorities of VMs whose pages are being de-duplicated. Therefore, a VM with high priority would find most of its de-duplicated pages on its local NUMA node.

Beside NUMA unawareness, we found that KSM scales poorly with the size of the memory. In hindsight, it is expected since KSM uses centralized data structures, like a single red-black tree to track all de-duplicated pages and one more for *all* candidates pages that may merge in the future. The time taken to access and update these structures increases with memory size, which delays de-duplication. Therefore, the responsiveness (time taken to remove duplicate pages) of KSM degrades on large memory systems.

For better responsiveness, nuKSM instead keeps two forests, instead of two trees. The number of trees in the forests is determined by the size of the memory in the system. For correct functioning, it is imperative for all identical pages to be assigned to the same tree for tracking. Therefore, nuKSM decides the tree that would track a page based on that page's checksum. Since two pages with the same content are bound to have the same checksum, nuKSM does not miss out on de-duplication opportunities.

In summary, we make the following contributions.

- We discover that NUMA-unawareness in Linux's KSM leads to significant unfairness and performance variability amongst concurrently running VMs. It could also lead to priority subversion where a higher priority process runs slower due to remote accesses induced by de-duplicated pages.

- We designed and implemented nuKSM in Linux that makes a judicious decision on the placement of the de-duplicated pages to reduce NUMA overhead and/or to ensure fairness and also avoid priority subversion.
- We also observed that overheads of KSM scale poorly with increasing memory footprint. We thus made nuKSM scale better by introducing a de-centralized approach to de-duplication.

A brief overview of memory deduplication and NUMA on the context of Linux/KVM is given in Chapter 2. Chapter 3 details the NUMA inplications of deduplication. Chapter 4 discuss the design of nuKSM. In Chapter 5, we evaluate nuKSM. Chapter 6 provides a brief overview of literature in the area of memory de-duplication. Finally, we conclude in Chapter 7.

Chapter 2

Background

In this section, we discuss memory de-duplication and NUMA in the context of Linux/KVM.

2.1 Memory de-duplication in Linux

It is not uncommon for memory pages to have duplicate contents, especially when multiple virtual machines are hosted on a single server. Virtual machines may be running the same OSes and a similar set of applications. The goal of de-duplication is to identify pages with identical contents and de-duplicate them to reduce the overall memory footprint. De-duplication involves mapping multiple virtual address ranges from the same or different VMs (processes) to one copy of the page. While many OSes and hypervisors support de-duplication [14, 34, 39], we focus our discussion around KSM in Linux/KVM. A process needs to register its virtual memory areas with KSM using the madvise system call with the MADV_MERGEABLE flag for it to be considered by KSM for de-duplication [14]. KVM automatically registers the entire memory of VMs to KSM for de-duplication.

Figure 2.1 shows an overview of the data structures used in KSM. The algorithm used to detect pages with duplicate content is important but often a resource-consuming building block of a memory de-duplication system. KSM maintains two red-black trees for detecting identical contents, namely stable and unstable. The tree nodes are arranged based on the content of the pages. All de-duplicated pages are placed in the stable tree, while all potential merging candidates are placed in the unstable tree. A page whose content has not been updated between the two most recent scans is considered a potential merging candidate. Frequently updated pages are unlikely to be de-duplicated with others. Further, the cost of identifying and de-duplicating a page is amortized only if it stays de-duplicated for a long duration. An update to a de-duplicated page leads to costly de-merging. Thus, KSM avoids considering those pages

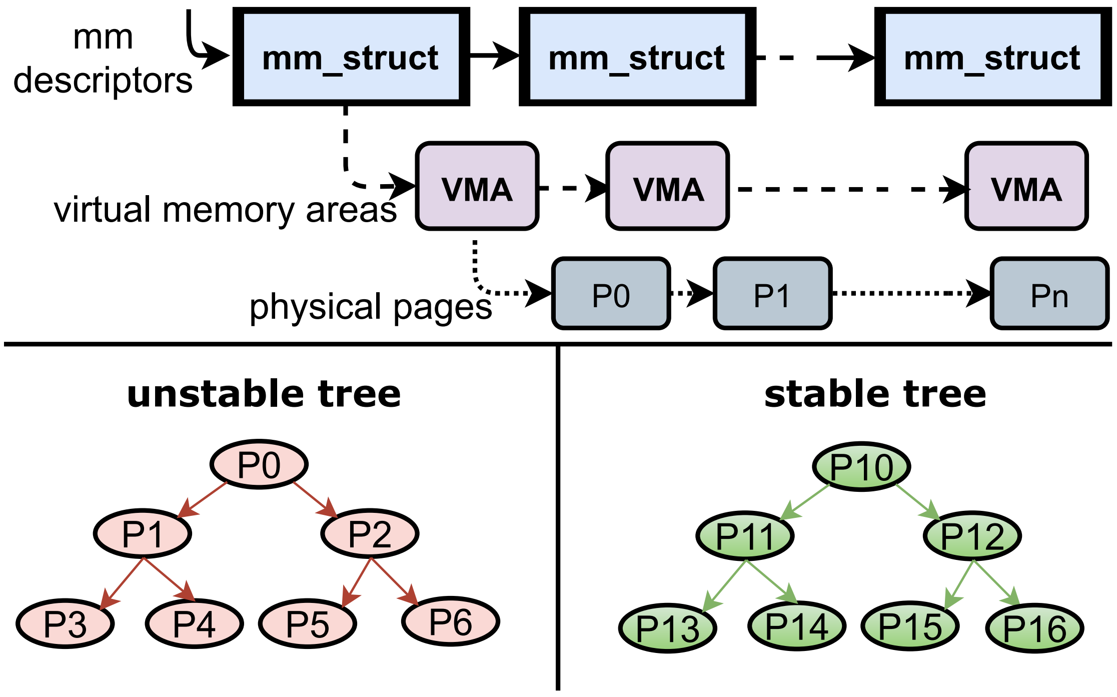


Figure 2.1: Key data structures involved in KSM. A mm_struct represents a virtual address space of a process. KSM chooses address spaces to scan from the list of registered mm_structs. A VMA represents a contiguous region in an address space. The unstable and stable trees are exclusively used by KSM for identifying candidates for de-duplication. P0, P1, ... , P16 are the mapped physical pages.

for de-duplication that are updated frequently. The stable tree is constructed once when KSM is initialized. However, the unstable tree is reconstructed from scratch in each scan by KSM to avoid unnecessary comparisons against recently updated pages. This is because pages in the unstable tree are not write-protected, and their content may have been updated between the scans.

KSM employs a kernel thread for ① finding opportunities for de-duplication and ② for performing de-duplication. Processes, including VMs, that are registered with KSM are maintained in a list (shown at the top of Figure 2.1). KSM's kernel thread wakes up periodically and starts scanning processes from this list one at a time. While scanning a process, KSM sequentially runs through the physical pages mapped to its virtual memory areas (represented by struct vm_area_struct) to identify candidates for de-duplication. The rate at which KSM scans pages to identify duplicate content (a.k.a., scan rate) is parameterized in Linux using two knobs - pages_to_scan and sleep_millisecs. KSM would sleep for sleep_millisecs milliseconds after scanning pages_to_scan pages. To control its CPU and memory bandwidth utilization, the thread scans a fixed (configurable) number of pages every fixed time unit. We refer to it as the scan rate of KSM. A scan of KSM is defined as a single traversal of all the virtual memory regions registered with it.

For each page, KSM performs a search in the stable tree to find if it is identical to one of the existing de-duplicated pages. On a match, the physical page being scanned is freed, and the virtual page that was mapping to the freed physical page is now mapped to the de-duplicated page found in the stable tree. The new mapping is rendered Copy-on-Write (COW), by unsetting the write permission bit in the corresponding page table entry in the page table of the scanned process, to prevent one process from modifying the content that is also mapped in other processes.

If there is no match in the stable tree, a checksum is computed on the contents of the page. It is then compared against the page's checksum that was computed by KSM during the last scan. KSM maintains reverse mapping to all the virtual addresses that maps to the physical page. Checksum from the previous scan is stored in the reverse mapping structure for the physical page frame. A mismatch between the two checksums signifies that the page has been modified, and thus, the page will not be considered for de-duplication in the current scan. Otherwise, pages of the unstable tree are searched for a match with the given page's content *i.e.,* compared with the already identified candidates for de-duplication. If there is a match, then de-duplication is performed as follows. The physical page frame in the unstable tree is freed. The virtual address range mapping onto the free physical page is then mapped to the page that is being scanned. As before, the mapping is rendered COW. The retained page is then added to the stable tree. If there is no match, the scanned page is added to the unstable tree. This page will then be compared with other candidate pages found during the current scan.

2.2 Non-uniform memory access (NUMA)

Modern servers often sport 2-4 sockets, with one CPU in each socket [1, 4, 38]. Each socket has local memory attached to it. Sockets themselves are connected via cache-coherent high-speed interconnects over the motherboard. All CPUs and all memory across the sockets are logically managed as a single system by the same OS image.

A CPU can access both local memory and memory attached to any other socket in the system *i.e.,* remote memory. However, latency to access remote memory is typically 1.5–2× higher than accessing local memory [1, 13]. Consequently, it gives rise to non-uniform memory access or NUMA.

A goal of OS's memory management is to avoid or hide overheads of accessing remote memory. For simplicity, we will refer to the overhead of accessing remote memory as *NUMA-Tax*. Linux enables tools like numactl and libnuma library that allows users/programmers to explicitly allocate memory from specific NUMA nodes and move the physical location of pages from one node to other in a NUMA system while the process is running via programming control [17]. Linux also supports automatic page migration via AutoNUMA [11]. AutoNUMA attempts to migrate pages across NUMA nodes at runtime to minimize NUMA-Tax. Importantly it does so without user intervention and induces page faults in regular intervals to identify local and remote accesses. By design, AutoNUMA migrates pages that are accessed frequently from a remote CPU and avoids migrating any pages that are accessed from multiple CPUs. Following the same principle, AutoNUMA does not attempt to migrate pages de-duplicated by KSM.

Chapter 3

NUMA implications of de-duplication

We discover that memory management's goal of better memory consolidation through aggressive de-duplication can conflict with its goal of keeping the NUMA-Tax low. This conflict leads to performance variability, unfairness, and priority subversion in the system.

The conflict arises since these objectives are *pursued in isolation* which leads to unintended implications on the overall system's behavior. For example, Linux's KSM enables high memory consolidation through de-duplication of pages across NUMA nodes. However, doing so can uncontrollably increase NUMA-Tax, if not careful. After a de-duplication, the accesses to the de-duplicated page would become remote for all VMs, except for the ones running on the node where it resides. We found that the NUMA-unawareness of Linux's KSM leads to avoidable and unequal distribution of NUMA-Tax across the VMs in Linux/KVM world.

It is possible to disable de-duplication across NUMA nodes, but that would hurt the goal of memory consolidation, particularly in large servers with several sockets. In an ideal world, one would thus allow de-duplication across nodes but expect memory management to contain the ill-effects of NUMA-Tax on system's behavior. In this work, we make progress toward this goal. However, we first analyze and quantify the ill-effects of NUMA-Tax induced by de-duplication in Linux/KVM.

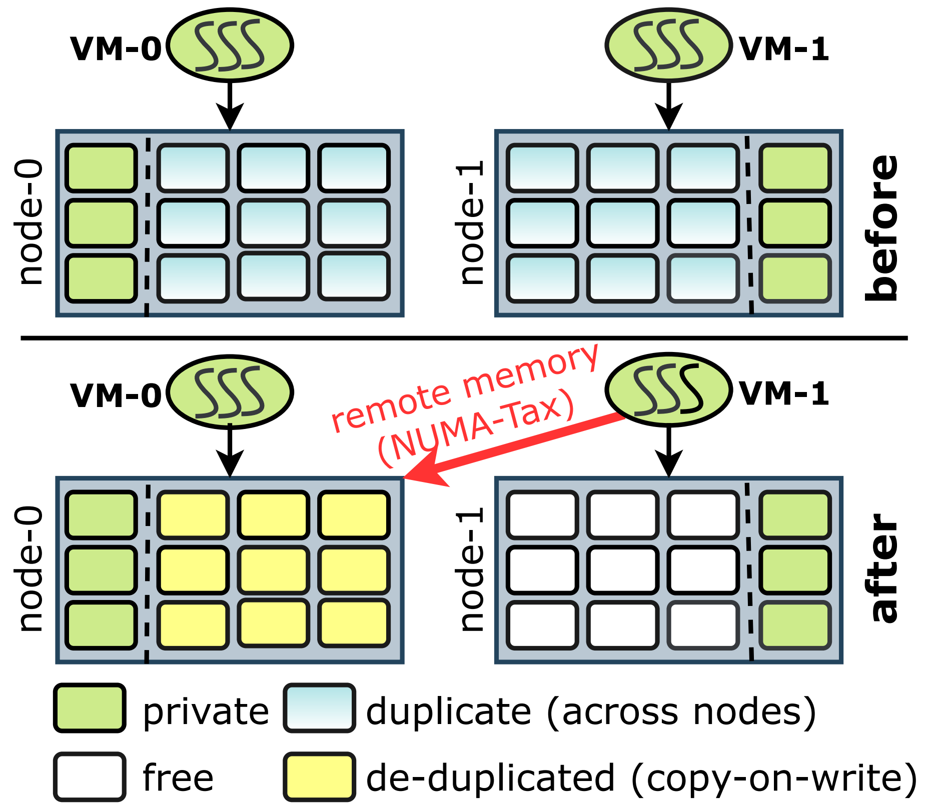
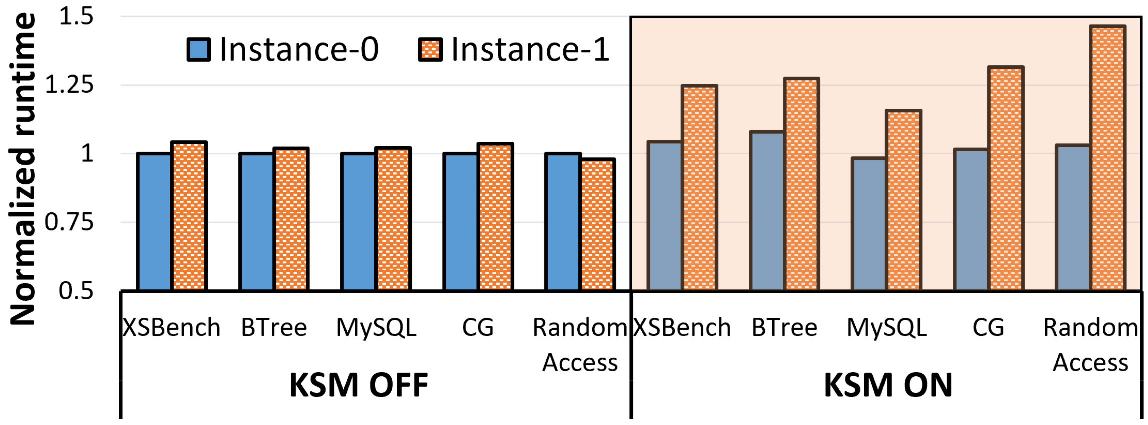


Figure 3.1: Effect of de-duplication on NUMA-Tax. Both VMs access local memory prior to de-duplication (top). After de-duplication (bottom), all merged pages are placed on node-0.

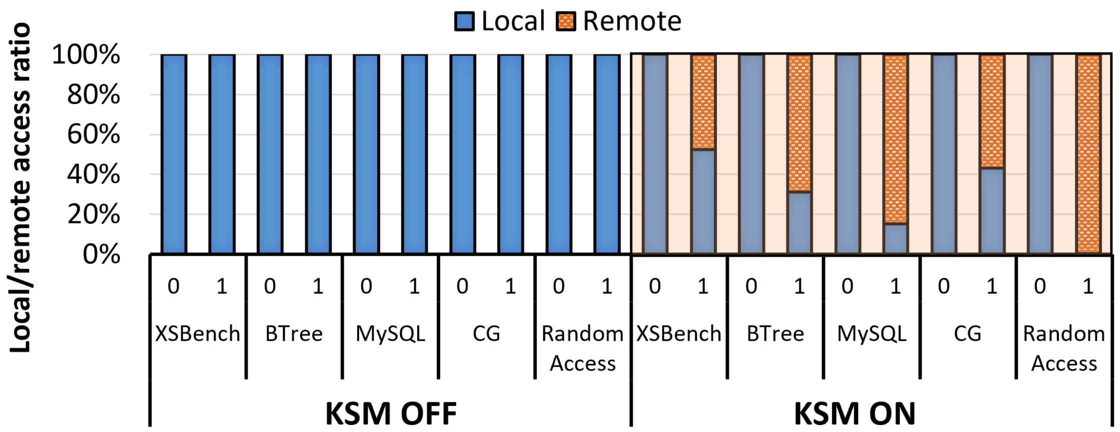
3.1 Performance variability and unfairness

Observation: *De-duplication across nodes unfairly penalizes some applications (VMs) due to NUMA-unaware placement of de-duplicated pages.*

We noticed that KSM usually places all de-duplicated pages on a single node. We root cause this behavior to the fact that KSM chooses where to place a de-duplicated physical page based on the order in which it scanned processes, oblivious of the underlying NUMA considerations. As discussed in Section 2.1, when contents of a page being scanned is identical to that of an existing candidate page in the unstable tree, KSM always chooses to retain the page being scanned and frees the one in the unstable tree. When the page being scanned matches with a page in the stable tree, the one from the tree is retained. In either case, *which page to retain and hence on which node is decided by the order in which* KSM *happens to scan address spaces*, disregarding potential NUMA implications. Further note that KSM scans the entire address space of a process at a time. Consequently, when two VMs contain many pages with identical content, then pages belonging to the VM that is scanned later are kept as de-duplicated pages while those of the first VM are released. This leads to an uncontrolled and unfair imposition of NUMA-Tax upon one (or more) of the VMs when they run on different nodes.



(a) Normalized execution time



(b) Local/remote memory access ratio

Figure 3.2: Execution time and local/remote memory access ratios of two identical instances of applications executing on VMs on different NUMA nodes. Execution times are normalized to that of Instance-0 without KSM.

We demonstrate this behavior with a simple example. Figure 3.1 shows a 2-socket system running two VMs with many identical read-mostly pages. The VMs are affined to separate NUMA nodes *i.e.,* VM-0 runs on node-0 and VM-1 runs on node-1. Suppose KSM scans VM-1's address space first, followed by VM-0's. In the first scan, KSM calculates the checksum of their pages, and both KSM trees remain empty at the end of the scan. In the next scan by KSM, pages of VM-1 are scanned, pages with similar content within VM-1's memory address space are merged and inserted into the stable tree. The pages which did not find a match for

	KSM OFF		KSM ON	
	Instance-0	Instance-1	Instance-0	Instance-1
XSBench	3252.993	3389.81	3395.643	4060.191
BTree	1466	1494	1582	1867
MySQL	934.9033	955.5301	919.0461	1080.877
CG	1072.5	1111.73	1089.94	1410.22
RandomAccess	99	97	102	145

Table 3.1: Execution time (in s) of two identical instances of applications executing on VMs on different NUMA nodes.

		Instance	Local Memory Accesses	Remote Memory Accesses
KSM OFF	XSBench	0	84733592881	1
		1	84743151656	991
	BTree	0	15853109912	0
		1	16004296183	410
	MySQL	0	28315885102	17
		1	30485193982	892
	CG	0	73121896749	0
		1	74148913028	792
	Random Access	0	770435510	25
		1	653502382	3
KSM ON	XSBench	0	85667248808	2300401
		1	43969443223	40036175337
	BTree	0	16446058547	5581317
		1	5186211551	11390879755
	MySQL	0	555469893	28687
		1	180097090	997854519
	CG	0	16933792847	2065423
		1	6671688629	8803273665
	Random Access	0	780137384	2826
		1	2331276	612051174

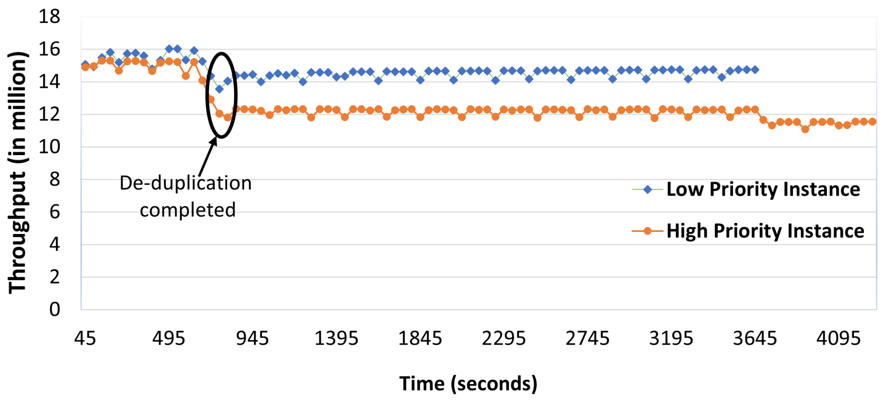
Table 3.2: Local/Remote Memory Accesses of two identical instances of applications executing on VMs on different NUMA nodes.

de-duplication are added to the unstable tree. In the same scan, pages of VM-0 are also scanned and searched for a match in the unstable tree. When some of these pages match with those of VM-1 in the unstable tree, the pages of VM-0 (which are placed on node-0) are retained as the de-duplicated pages. This way, all de-duplicated pages get consolidated on node-0. Consequently, VM-1 experiences high and an unfair share of NUMA-Tax after de-duplication, while VM-0 continues to find most accesses to be local.

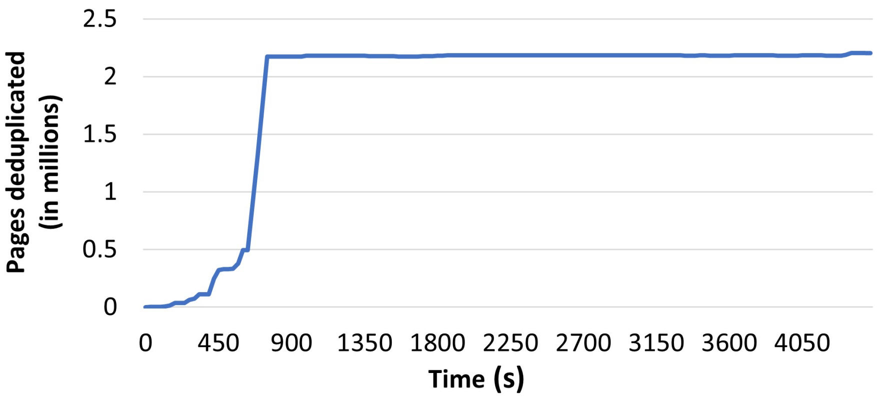
Empirical analysis: We quantify the aforementioned ill-effects of KSM on NUMA-Tax over a set of applications on a two-socket server (Section 5.1 details methodology). We set up two virtual machines VM-0 and VM-1, each executing an instance of the same application simultaneously *i.e.,* VM-0 executes Instance-0 while VM-1 executes Instance-1. We bind the VMs to different NUMA nodes and instantiate VM-1 one minute after VM-0. Table 3.1 reports the execution time of each applications. Figure 3.2a reports the performance of each application. The performance of each instance is normalized to the runtime of Instance-0 of the same application with KSM off.

When KSM is disabled, the performance of both instances is similar, *i.e.,* no unfairness. Since instances run on different nodes, there is little CPU and memory interference. However, when KSM is enabled, we observe a significant performance difference of 16%-46% between the instances of the applications. For example, RandomAccess and CG in VM-1 slow down by 46% and 31% compared to when KSM was disabled while their performance is largely unaffected in VM-0. *Importantly,* Instance-1 *slows down significantly in all cases.*

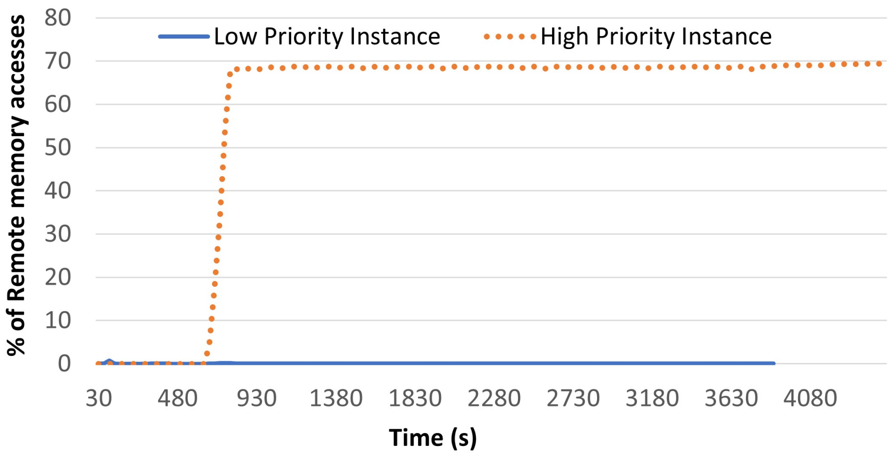
Applications in VM-1 get unfairly slowed down because VM-1's pages first get added to the unstable tree, and then VM-0's pages are retained on node-0 after de-duplication. Table 3.2 shows the application's memory (DRAM) accesses (*i.e.,* after missing in the cache). Figure 3.2b shows the breakdown of application's memory accesses into local and remote memory. We observe that when KSM is enabled, Instance-1 suffers from a high percentage of remote memory accesses while Instance-0 enjoys local accesses. The large performance gap observed in Figure 3.2a is a direct consequence of these high latency remote accesses experienced by Instance-1.



(a) Throughput



(b) Pages de-duplicated



(c) % of remote memory access

Figure 3.3: Throughput, pages de-duplicated and percentage of remote memory accesses of two identical instances of BTree running with different priorities with KSM.

3.2 Priority subversion

Observation: *De-duplication subverts user's priority goals.*

The fact that KSM retains a copy of a page based on the order in which it scans address spaces makes it vulnerable to priority subversion as well. Priority subversion is the circumstance where a higher priority process gets penalized by a low priority process due to priority-unaware resource allocation [29].

Empirical analysis: To demonstrate an example of priority subversion due to KSM, we execute two instances of an application BTree as in the previous subsection. In addition, we assign different priorities to the VMs. VM-0 runs with lowest priority with its nice value set to 20, and VM-1 runs with highest priority whose nice value is set to -20.

Figure 3.3a shows the effect of de-duplication by KSM on the throughput. We report throughput as the number of random lookups in B+ tree in 45 seconds interval. While both instances start with similar throughput, that of the high priority instance quickly drops by more than 15% as memory gets de-duplicated. Consequently, it also takes longer to complete. Evolution of de-duplicated pages with respect to time is being shown in Figure 3.3b. We can also observe that, as pages de-duplicate, the percentage of remote memory accesses increases(as shown in Figure 3.3c). The priority subversion is also a side-effect of KSM's behavior where scanning order determines which nodes get to keep the de-duplicated pages.

Priority subversion is an unintended consequence of memory de-duplication on NUMA platforms. Unfortunately, there is no way in Linux today for users to ensure that the notion of priority is honored. While disabling de-duplication across nodes is one option to avoid this problem, it gives up a significant opportunity for memory consolidation.

3.3 Low responsiveness with large memory

Observation: *KSM scales poorly with increasing memory size.*

Unrelated to NUMA, we further noticed that KSM does not scale well to large memory systems. This is primarily due to the centralized nature of KSM's page comparison. It maintains one set of trees for the entire memory. As memory size grows, so grows the height of the trees. While scanning processes, KSM compares each page with stable tree nodes and then (if needed) with unstable tree nodes to identify duplicate contents. The number of comparisons grows with the height of the tree, which, in turn, grows with the memory size.

To demonstrate this, we created a micro-benchmark which generates 2X (X = 1024, 10240, 102400, 1024000) number of pages with pairwise duplicate contents. Thus, after merging, we would have X number of nodes in the stable tree. After the merging is complete, we run a

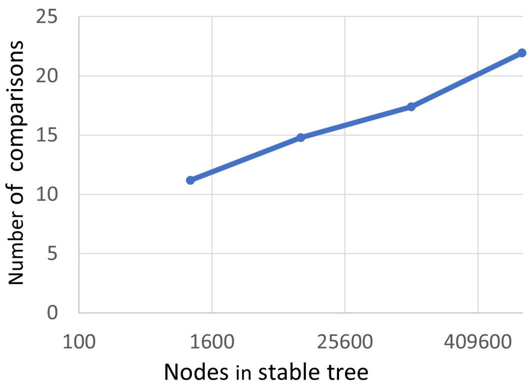


Figure 3.4: Average Number of comparisons with different number of nodes in the stable tree.

workload that will not have any of its page content similar to the pages being generated by the previous micro-benchmark. Figure 3.4 reports the average number of comparisons in the stable tree with different values of X. The number of comparisons in the stable/unstable tree increases *logarithmically* with the number of nodes in them. This is primarily a side-effect of the increasing size of the red-black trees that are used to maintain and identify de-duplicated pages (which are part of the stable tree) and the potential merging candidates (that reside in the unstable trees). As discussed in Section 2.1, while scanning virtual memory regions, pages are compared first in the stable tree and then (if needed) in the unstable tree. Therefore, the overall cost of scanning depends on the height of these trees, which increases logarithmically with the number of pages.

As we will later show in Chapter 5, KSM fails to de-duplicate memory quickly enough and runs into Out-of-Memory (OOM) errors when the memory size increases to hundreds of GiBs. In other words, if KSM was more responsive, OOM could have been avoided.

Chapter 4

Design and implementation

In this chapter, we discuss the design of nuKSM. Our design objectives include ① minimizing performance variability and unfairness due to memory de-duplication across NUMA nodes, ② ability to distribute NUMA-Tax based on the priority of different processes and avoiding priority subversion, and ③, finally, improving the responsiveness of de-duplication in systems with large memory. We will describe how nuKSM achieves each of these objectives. We implement nuKSM in Linux kernel version 5.4.0 by extending KSM.

4.1 Addressing performance variability and unfairness

nuKSM first strives to avoid paying the NUMA-Tax by judiciously keeping a de-duplicated page on a NUMA node that is expected to access the de-duplicated page often. This is driven by the observation that an application pays the NUMA-Tax only when accessing a page on a remote node. An application would observe little impact of NUMA-Tax on its performance if one of its infrequently accessed pages is de-duplicated and kept on a remote node. If it is not immediately discernible which of the accessing application/VM is likely to access a de-duplicated page more often, nuKSM evenly distributes the NUMA-Tax among applications/VMs. This policy is key to avoid unfairness in execution, and also avoid paying NUMA-Tax when possible.

We provide a simple example to illustrate this policy in working. Let us consider two virtual machines VM-0 and VM-1, that are running on separate NUMA nodes and sharing five pages P0, P1, P2, P3, and P4. Let us also assume that P0 is more frequently accessed by VM-0 and P1 is more frequently accessed by VM-1. Further, let us also assume that P2 and P3 are accessed by both VMs with similar frequency, and P4 is mostly inactive. For this example, nuKSM would place P0 close to VM-0, P1 close to VM-1. For an even distribution of NUMA-Tax, nuKSM would place one of P2 & P3 close to one VM and the other close to the other VM.

Since P4 is mostly inactive, its placement is not critical for performance, and nuKSM could place it in either of the nodes. This way, two out of the four active pages will be local to each VM, and NUMA-Tax is evenly distributed for the rest.

An implementation of the above policy would require measuring access frequency of pages to be de-duplicated. Typically, this information is obtained from page table access bits, which are set by the hardware on an access to corresponding pages. The OS can periodically clear access bits in page tables and check them after a certain time period to find which pages are being accessed frequently [23, 28, 42]. However, doing so would add extra overhead to KSM. Further, prior works indicate that this technique is expensive on large memory systems due to the high CPU overhead involved in traversing the page tables for clearing and reading the access bits [23, 42].

In nuKSM, we instead leverage the information *already available* to the page reclamation subsystem of Linux. The page reclamation algorithm is a variant of the well-known clock algorithm [6], which maintains two bits per page, namely accessed and referenced. Based on the value of these bits, pages are divided across two lists active and inactive. Pages that are infrequently accessed are accumulated in the inactive list while the rest of the pages find their place in the active list. Under memory pressure, pages from the inactive list are swapped out to storage. Implementation of the page reclamation subsystem and its heuristics have been heavily optimized over the years by the Linux community [33]. We thus piggyback upon the hints about a page's access frequency already available from the page reclamation subsystem to realize nuKSM's policy of which copy of pages with identical content to retain.

To state our approach succinctly, while de-duplicating two pages across nodes – say P0 from node-0 and P1 from node-1 – we first check which of the pages are frequently accessed, *i.e.,* part of the active list. If only one of them is active (say P0), we use it as the final de-duplicated page and free the other page (P1). If both P0 and P1 are in active list, nuKSM uses round-robin policy to determine which one to retain and which one to free. For example, if the first of two active pages is placed on node-0 while de-duplicating, the next one would be placed on node-1, and so on. This simple policy is enough to ensure that the set of frequently accessed pages is evenly distributed across nodes. Finally, if both pages are inactive, they are also distributed evenly across nodes using round-robin – this helps in balancing memory allocation to avoid thrashing a particular node. However, there is typically no performance implication of the placement of inactive pages.

4.2 Priority based memory de-duplication

Fairness and performance predictability are not always the most important objective in some execution environments. For such cases, nuKSM enables users to configure which process should enjoy more (or less) remote memory accesses based on their priorities. If priority-based memory de-duplication is enabled, nuKSM tries to distribute NUMA-Tax in the same ratio as the relative priority of processes that share the de-duplicated pages – unlike in KSM, where the distribution of NUMA-Tax is arbitrary.

Instead of introducing a new priority scheme, nuKSM inherits Linux's process priority *i.e.,* nice values. The nice value is an integer between -20 (highest priority) to 20 (lowest priority). It indicates relative priorities of different processes that form the basis of CPU sharing in implementing a scheduling policy. nuKSM repurposes the nice values while de-duplicating pages under this policy. For simplicity, we add 21 to each nice value to convert them to non-zero positive integers; we refer to these scaled values as snice.

When de-duplicating pages, we first calculate *nuShare*– a positive real number between 0 and 1 – that captures the preference of the current process whose page is being scanned, relative to all processes with whom it would share the de-duplicated page. *nuShare* of a page p is calculated using snice values as follows:

$$nuShare(p) = 1 - \frac{\mathsf{snice}(current)}{\sum_{\forall\ task\ using\ p} \mathsf{snice}(task)}$$

A high value of *nuShare* represents a stronger preference of making the de-duplicated page local to the process that is currently being scanned. The *nuShare* is then compared against a pseudo-random number generated in the range between 0 and 1. If the value of *nuShare* is larger than the random number, then the page being scanned is retained as the de-duplicated copy. This ensures that access to the de-duplicated page from the scanned process is not penalized because of de-duplication. Otherwise, we use the other page (from either the stable or the unstable tree) as the de-duplicated page and free the current page. This strategy helps in ensuring that the distribution of de-duplicated pages across NUMA nodes converges to the ratio of priority of different processes when many pages are de-duplicated.

4.3 Enhancing responsiveness

Independent of nuKSM's primary goal of making de-duplication NUMA-aware, nuKSM also attempts to scale de-duplication better to large memory systems. As observed earlier, KSM's

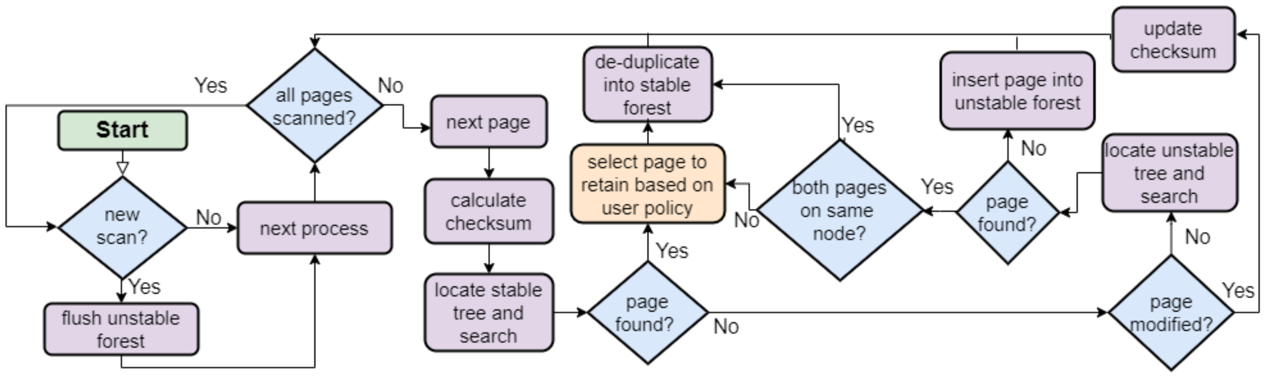


Figure 4.1: Memory de-duplication workflow in nuKSM.

low responsiveness is rooted in having one set of trees for covering the entire memory.

In nuKSM, instead of using one stable and one unstable tree, we use two forests *i.e.,* many stable and many unstable trees, represented as an array of trees. The index of a page in that array is a function of the checksum of that page's content (*i.e., index = page_checksum(page) % number of trees*). Note that two pages with identical contents will have the same checksum and, thus, index into the same stable and unstable tree. Hence, nuKSM does not miss any de-duplication opportunity.

Using checksum-based indexing into the array of trees allows distributing pages across many sets of trees. Consequently, it helps in limiting the height of each tree, which in turn reduces the number of comparisons required while searching for a match in a tree. In other words, this approach automatically reduces the number of unnecessary page comparisons because two pages are never compared if they index into different trees.

Using a forest is a scalable design since the number of trees in the forest can be adjusted based on the size of physical memory. For example, if memory size is doubled, doubling the number of trees would ensure that the average height of a tree does not increase, and thus, the number of comparisons remains similar. However, unnecessarily using a very large number of trees can introduce overhead, especially because the unstable trees are flushed and reconstructed from scratch in each scan. We empirically found that using one stable and one unstable trees per 100 MiB memory provides a reasonable balance between the cost vs. benefits of using a de-centralized forest-based approach.

4.4 Putting it all together

Finally, we depict the entire workflow of nuKSM with Figure 4.1. nuKSM starts by periodically scanning address spaces from the list of processes that have registered with it. The VMs, *i.e.,* KVM processes register their entire memory by default. The unstable trees are flush prior to starting a new scan. For each page being scanned, it is indexed into the forest of stable and unstable trees, based on its checksum. The corresponding stable and unstable trees are then searched for merging opportunities. When there is a match, the decision on which page to retain and which one to free depends upon user settings. By default, nuKSM uses the principle described in Section 4.1 that ensures equitable distribution of NUMA-Tax for fairness. However, it can be changed to the one based on priority (Section 4.2), with a *sysfs* configuration knob. The de-duplicated page is added to the stable forest. If the page does not match in either trees, it is added to the unstable forest.

4.4.1 Scaling to many sockets

While we limited our discussion so far to only two sockets for the ease of exposition, nuKSM's all three design aspects seamlessly extend beyond two sockets. First note that the priority-based memory de-duplication (Section 4.2) is agnostic to the number of sockets. Its calculation of *nuShare* that dictates distribution of NUMA-Tax is unaffected by number of sockets. Similarly, the number of trees for improving de-duplication's responsiveness (Section 4.3) is determined solely by the amount of physical memory.

That leaves us to discuss how nuKSM's algorithm for fairness (Section 4.1) scales to many sockets. Let us consider N processes, each in its own VM, are running on K sockets. Let us also assume that each of those N processes has a page with the same content that nuKSM would de-duplicate. Now, note that like KSM, nuKSM considers only two candidate pages for de-duplication at a time. A candidate page may already be a de-duplicated copy itself. Let us consider that at a given time nuKSM has already de-duplicated N-1 pages with duplicate contents from K-1 nodes onto a single de-duplicated page, say P_x. Now suppose that nuKSM finds the candidate page for de-duplication, P_y, having the same content as P_x and is currently placed on node K.

nuKSM should decide which one of these two copies to retain based on the same principle of minimizing the expected NUMA-Tax. Specifically, nuKSM considers three conditions. ① Both candidate pages are in active lists of their respective NUMA nodes, ② only one of the page is in active list, and ③ both the pages are in inactive lists. Under the first condition, *i.e.,* when both P_x and P_y are in active list, nuKSM tries to evenly distribute the de-duplicated pages across NUMA nodes where the original pages resided before de-duplication. To achieve this, nuKSM retains the page P_y with probability p, where $p = 1/K$. nuKSM generates a pseudo-random number between 0 and 1. If it is smaller than p, then the page P_y is retained and P_x is freed. Otherwise, nuKSM does the opposite. Under the second condition, nuKSM keeps the page that is in active list while freeing the other, as usual. If both pages are in inactive list then there is no expected performance implications of NUMA placement. Still, nuKSM uses the same technique as used for the first condition, to evenly distribute the de-duplicated pages across NUMA nodes.

Chapter 5

Evaluation

We evaluate nuKSM to answer the following questions: (1) how does nuKSM's NUMA-aware memory de-duplication perform with respect to fairness and performance variations? (2) how nuKSM's priority-based de-duplication helps users in controlling the distribution of NUMA-Tax? and (3) how responsive is nuKSM in exploiting de-duplication opportunities in large memory systems?

5.1 Methodology

We conduct all measurements on a dual-socket Intel Xeon Gold 6140 (Skylake) server with 18 cores and 192 GiB DDR4 physical memory per socket. The processor runs at a base frequency of 2.30 GHz with a 25MiB L3 cache. We disable the turbo boost and hyperthreading to minimize performance variations. We use Linux v5.4.0 as the kernel running in an Ubuntu18.04 guest OS, and the same as the host with KVM hypervisor. We use Linux v5.4.0-nuKSM as the modified de-duplication system. Both KSM and nuKSM operate at the same rate, scanning 1K pages before sleeping for 100 milliseconds. Each virtual machine is configured, using libvirt, with four vCPUs and 30 GiB memory, unless specified otherwise. To execute a VM on a specific socket, we bind its memory allocation to that socket, in addition to pinning its virtual CPUs to the physical CPUs of that socket. In all experiments, VM-0 runs on node-0 and executes Instance-0 of the applications, while VM-1 runs on node-1 and executes Instance-1. Our evaluation focuses on a mix of real-world databases and high-performance computing applications, and memory-intensive micro-benchmarks that are sensitive to NUMA-Tax. Table 5.1 provides further details of our evaluation platform and workloads.

Hardware platform	
Model	2-socket Intel Xeon Gold 6140
CPU cores	18 cores per socket @ 2.30GHz
Cache	25MiB shared L3 cache
Memory	DDR4-2666, 192GiB per socket Latency (in ns): 89 (local), 139 (remote) Bandwidth (GiB ps): 110 (local), 51 (remote)
Benchmarks	
XSBench [37]	A mini-app representing a key computation kernel of the Monte Carlo neutron transport algorithm memory footprint: 11 GiB, thread count: 4
MySQL [10]	A popular database service, benchmarked with 100 sysbench clients using in-memory tables memory footprint: 20 GiB, thread count: 1
BTree [31]	Random lookups in a B+ tree memory footprint: 5.6 GiB, thread count: 1
RandomAccess	Random lookups in a large array memory footprint: 2.8 GiB, thread count: 1
CG [3]	Implementation of congruent gradient algorithm memory footprint: 3.5 GiB, thread count: 4

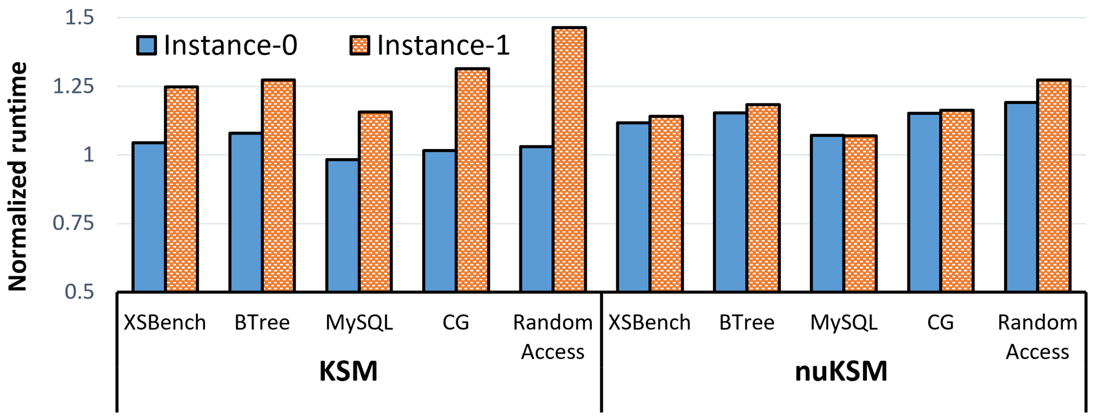
Table 5.1: Details of the evaluation platform and benchmarks.

5.2 Memory de-duplication for fairness

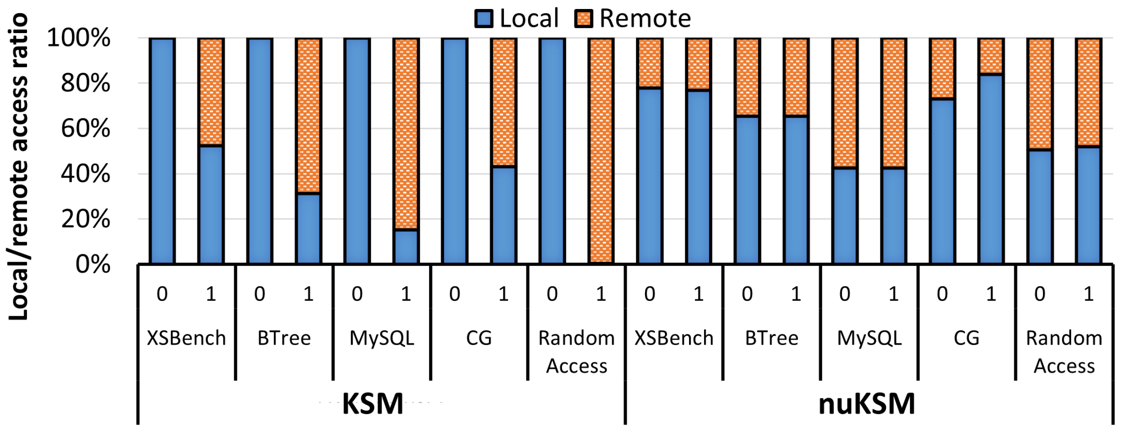
We first evaluate how nuKSM's NUMA-awareness helps in moderating arbitrary performance variability and ensures fairness among co-running VMs. We conduct an experiment similar to the one discussed in Section 3.1 where two VMs running identical applications are placed on different sockets (nodes). Figure 5.1 shows the result of our experiments.

KSM introduces high performance variability and thus, unfairness among applications running on different VMs (as discussed in Section 3.1), ranging from 15% performance difference between two instances of MySQL to 46% for those of RandomAccess. In contrast, the difference is almost negligible in nuKSM, on average, and maximum 4% for RandomAccess as shown in Figure 5.1a.

Figure 5.1b shows the percentages of local and remote memory access for both instances of each application, with KSM and nuKSM, respectively. As we discussed in Section 3.1, different instances of an application witness different amounts of remote memory accesses under KSM. However, with nuKSM, we observe that the variability of remote access percentages are quite less across both the instances for every application. This confirms that nuKSM distributes NUMA-



(a) Runtime normalized to that of Instance-0 when KSM is disabled



(b) Local/remote memory access ratio

Figure 5.1: Performance, fairness and local/remote memory accesses ratio of two identical instances of different applications with KSM and nuKSM.

Tax fairly, unlike KSM, and helps avoid performance variability and unfairness in application performance.

We quantify fairness (or lack thereof), using a well-known metric that is used to measure performance in multi-programmed workloads [16]. For two instances of an applications *I0* and *I1*, fairness is calculated as follows:

$$fairness(I0,\ I1) = \frac{min(slowdown(I0),\ slowdown(I1)}{max(slowdown(I0),\ slowdown(I1)}$$

For N instances of an application *I0*, *I1*, ... , *IN*, fairness is calculated as follows:

$$fairness(I0,\ I1, ...,\ IN) = \frac{min(slowdown(I0),\ slowdown(I1), ...,\ slowdown(IN))}{max(slowdown(I0),\ slowdown(I1), ...,\ slowdown(IN))}$$

The *slowdown* is measured with respect to the baseline system. In our case, the baseline represents the case where de-deduplication (KSM) is disabled. Note that the value of fairness lies between 0 and 1. A higher value of fairness is desirable as it signifies low-performance variation.

Benchmark	**fairness**		**Normalized combined**	**memory saved (GiB)**	
	KSM	nuKSM	**runtime of nuKSM**	KSM	nuKSM
XSBench	0.84	0.98	0.99	10.76	10.79
BTree	0.85	0.98	0.99	5.18	5.29
MySQL	0.85	0.99	1.00	15.90	15.97
CG	0.77	0.99	0.99	2.40	2.39
Random-Access	0.70	0.94	0.99	3.14	3.16

Table 5.2: Amount of de-duplicated memory and fairness with KSM and nuKSM. Rightmost column shows the combined performance of nuKSM, normalized to KSM.

Table 5.2 shows fairness in KSM and nuKSM. nuKSM is close to an ideal system as the value of fairness is very close to 1 in all cases. Specifically, nuKSM improves fairness from 0.84 to 0.98 for XSBench, 0.85 to 0.98 for BTree, 0.85 to 0.99 for MySQL, and from 0.77 to 0.99 for CG.

Note that nuKSM improves fairness at the cost of some performance loss of Instance-0 since it distributes a portion of NUMA-Tax to it, instead of only burdening Instance-1. However, the performance of Instance-1 improves significantly. A keen reader may wonder if relative degradation in the performance of Instance-0 outweighs the gain of Instance-1. We therefore also show the normalized combined runtime of nuKSM for each application. The combined runtime is calculated by adding the total execution time of both instances of an application. For normalization, the combined runtime in nuKSM is then divided by that of the same application in KSM. Normalization helps discard instance-specific runtime differences and provides a measure of overall system throughput. Table 5.2 shows that the normalized combined performance of nuKSM is similar to that of KSM. It confirms that there is no overall performance loss in nuKSM. In summary, nuKSM improves fairness significantly while achieving the same overall performance as KSM.

Finally, one may also wonder whether nuKSM was effective in saving memory – the primary objective of de-duplication. In the last set of sub-columns of Table 5.2, we report memory saved by KSM and nuKSM. Clearly, nuKSM is at least as effective as KSM in saving memory, while also ensuring fairness.

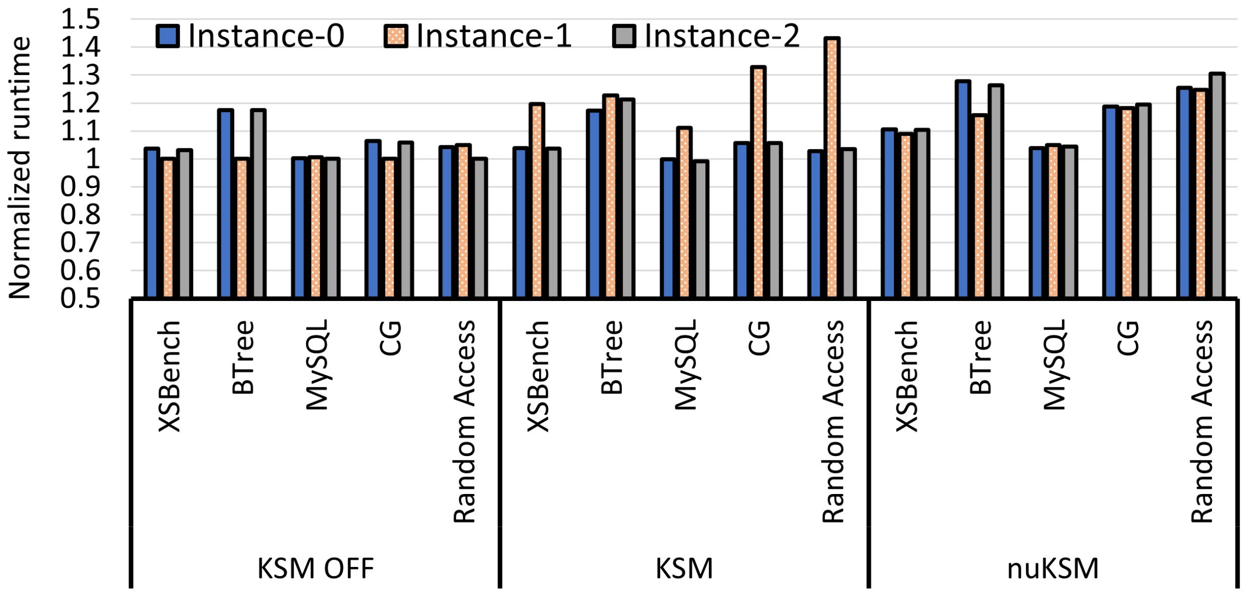


Figure 5.2: Performance of three identical instances of different applications with KSM off, KSM and nuKSM. Execution time is normalized to the runtime of Instance with minimum runtime with KSM OFF.

5.2.1 Extending beyond two VMs/processes

We also evaluate how nuKSM scales beyond two VMs/processes. We conducted experiments with three VMs across two nodes of our server, with VM-0 and VM-2 executing on node-0, and VM-1 executing on node-1. VM-0 executes Instance-0, VM-1 executes Instance-1 and VM-2 executes Instance-2 of the applications. Figure 5.2 shows that NUMA leads to significant performance variability and unfairness under KSM and a very little under nuKSM. We can see that, almost for each application we could see performance degradation in Instance-1, which runs on node-1. Instance-0 and Instance-2 performs quite equally. We could find a particular outlier in BTree. We observed that under KSM, the three BTree instances performed roughly equal with very little performance variations, while for nuKSM, Instance-1 performed better than Instance-0 and Instance-2. We thought this could be due to congestion on node-0 as both the VMs (VM-0 and VM-2) are being run on node-0. So we also conducted an experiment with four VMs for BTree across two nodes of our server. VM-0 and VM-2 runs on node-0 and VM-1 and VM-3 runs on node-1. VM-0 executes Instance-0, VM-1 executes Instance-1, VM-2 executes Instance-2 and VM-3 executes Instance-3 of the application. Figure 5.3 shows the performance of four identical instances of BTree with KSM and nuKSM. We can see that there is significant performance variation under KSM but very little under nuKSM. In summary, we demonstrate that nuKSM is able to eliminate performance overheads due to NUMA even when more than two VMs are involved.

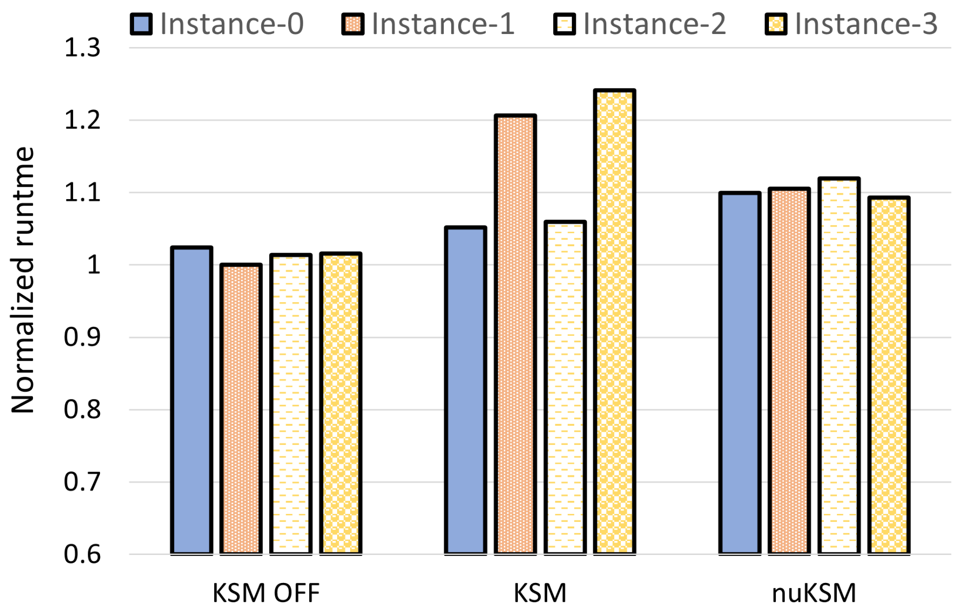


Figure 5.3: Performance of four identical instances of BTree with KSM off, KSM and nuKSM. Execution time is normalized to the runtime of Instance with minimum runtime with KSM OFF.

5.3 Priority based memory de-duplication

In Section 3.2, we demonstrated how KSM in Linux/KVM subverts priority goals with users having no control over how NUMA-Tax is distributed. Here, we show how nuKSM enables users to adjust the distribution of NUMA-Tax at a fine grain.

We create five different configurations based on the priorities of two VMs, as shown in Table 5.3. The table also shows the fraction of de-duplicated pages that are local to each VM after nuKSM has de-duplicated all identical pages. Each configuration is represented as C-P0:P1 where P0 denotes the relative priority of VM-0 against the priority of VM-1 (*i.e.*, P1). nuKSM places de-duplicated pages in the same ratio as the relative priority of the VMs. For example, in configuration C-10:1, out of every 11 de-duplicated pages, 10 pages are placed on node-0 while one page is placed on node-1.

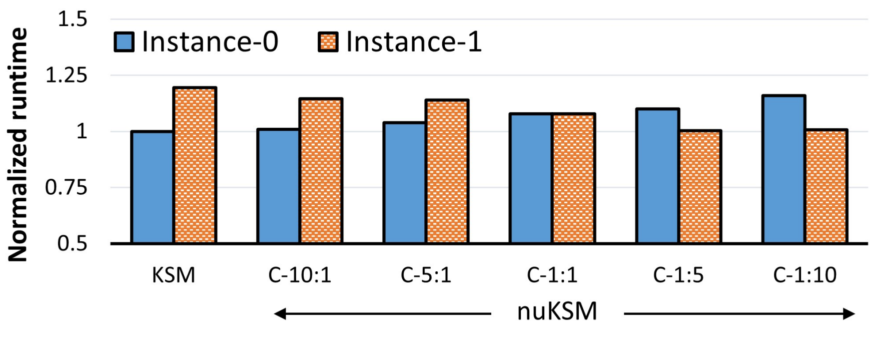
Figure 5.4 shows our experiments for three applications BTree, XSBench and MySQL for all five priority combinations in nuKSM. All configurations lead to similar performance in KSM since it is oblivious of process priorities. Hence, KSM is shown once for this experiment.

The relative priority of VM-0 decreases from left to right in each sub-figure of Figure 5.4. Consequently, the fraction of de-duplicated pages that is local to Instance-0 also decreases from left to right *i.e.,* from 91% in C-10:1 to 50% in C-1:1, and further to only 9% in C-1:10. At the same time, the fraction of de-duplicated pages that is local to Instance-1 increases from left to right. As expected, the runtime of applications decreases when they receive more local memory. For example, the runtime of Instance-0 of BTree, XSBench and MySQL is 14%, 13% and 12% lower than that of Instance-1 in configuration C-10:1 but higher by a similar margin when their relative priorities are inverted in configuration C-1:10.

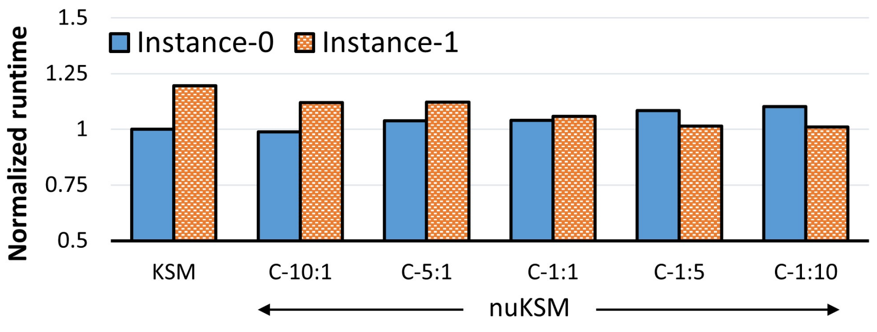
config.	**VM-0**		**VM-1**		**% de-duplicated pages**	
	nice	snice	nice	snice	VM-0	VM-1
C-10:1	-20	1	-11	10	91%	9%
C-5:1	-20	1	-16	5	83%	17%
C-1:1	-20	1	-20	1	50%	50%
C-1:5	-16	5	-20	1	17%	83%
C-1:10	-11	10	-20	1	9%	91%

Table 5.3: Different priority configurations based on the nice values of VMs and the expected fraction of de-duplicated pages local to each VM in the corresponding configuration.

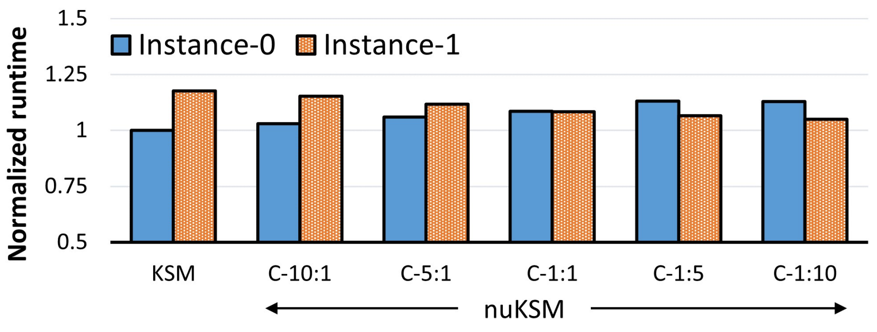
Overall, Figure 5.4 shows that nuKSM can distribute NUMA-Tax accurately and at a fine grain based on relative priorities assigned by the user. Note that both instances perform roughly similar in C-1:1. This configuration represents a special case of priority-based de-duplication wherein both VMs run with the same priority, and hence nuKSM ensures fairness.



(a) BTree



(b) XSBench

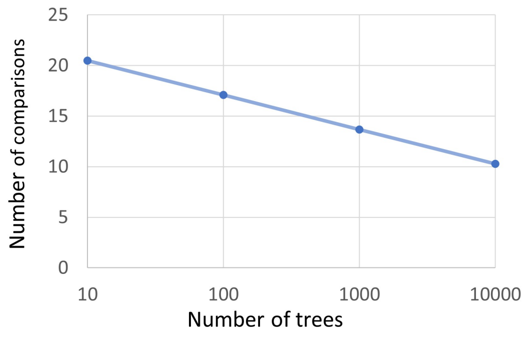


(c) MySQL

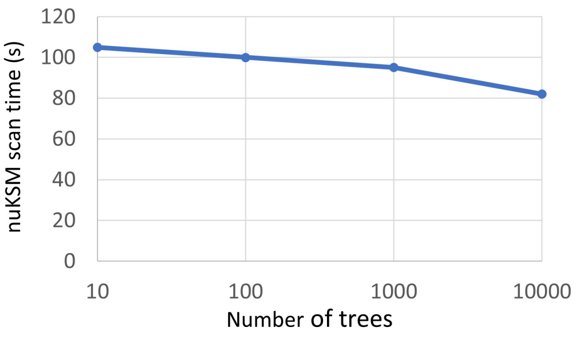
Figure 5.4: Execution time of two instances of different applications executing on separate NUMA nodes with different priorities in KSM and nuKSM. Execution time is normalized to the runtime of Instance-0 with KSM. Performance in KSM is unaffected by priority and hence it is shown once.

5.4 Responsiveness with large memory

We now demonstrate the effect of nuKSM's de-centralized design of memory de-duplication. We conduct the same experimental studies as described in Section 4.3. We now experiment with varying the number of trees in the stable/unstable forests in nuKSM to show the effect of a varying number of trees in the number of comparison in the stable tree and consequently the scan time of nuKSM. Figure 5.5a shows the number of comparisons in stable tree per page with varying number of trees. Figure 5.5b shows the scan time of KSM for a round with varying number of trees. We can see that number of comparisons and scan time reduces with an increasing number of trees in the forest.



(a) Number of comparisons in stable tree per page



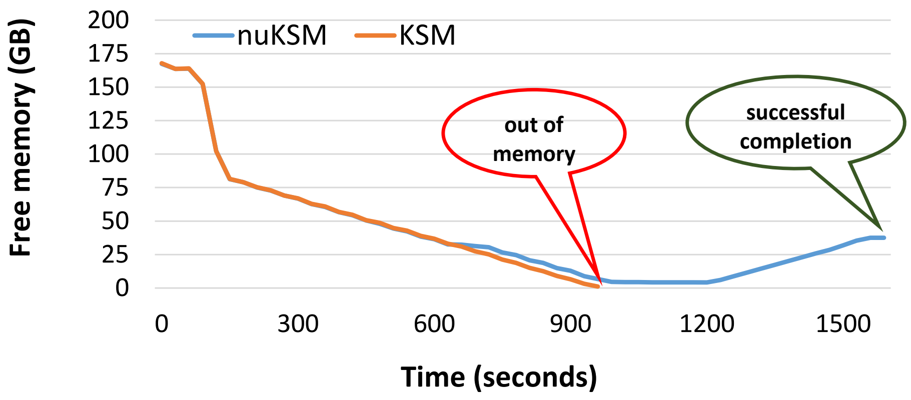
(b) nuKSM scan time

Figure 5.5: Number of comparisons and ksm scan time with varying number of trees in stable/unstable forest in nuKSM

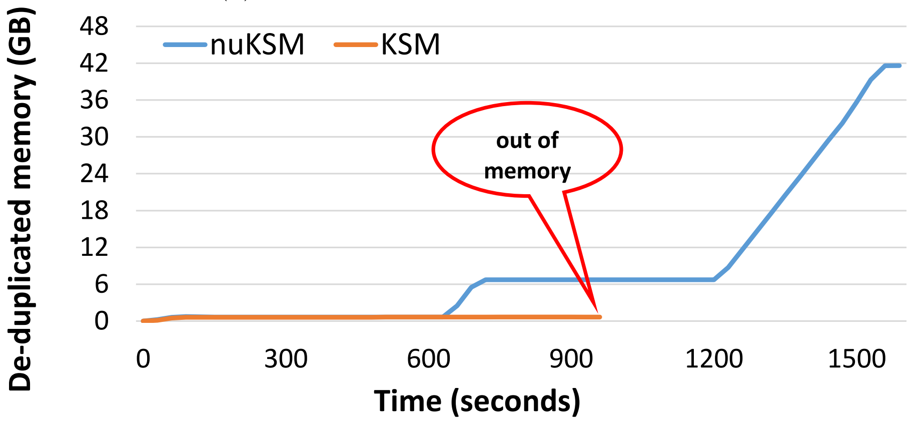
We now demonstrate the effect of nuKSM's de-centralized design in improving the responsiveness of memory de-duplication when hundreds of GBs of memory is in use.

We run two 40 GiB instances of XSBench, and a background job that allocates 2 GiB physical memory every 15 seconds. The background job allocates total 100 GiB memory and registers itself for de-duplication. The background job simulates the effect of progressively increasing memory pressure. All the workloads run on node-0 to avoid NUMA effects. The node has about 180 GiB memory available. The combined memory footprint of all three processes is slightly higher than the memory available. Hence, the system will run out of memory if de-duplication does not free memory fast enough,*i.e.,* if not responsive enough. To adjust to larger memory size, we also configure the scan rate to 10K pages every 100 milliseconds in both KSM and nuKSM.

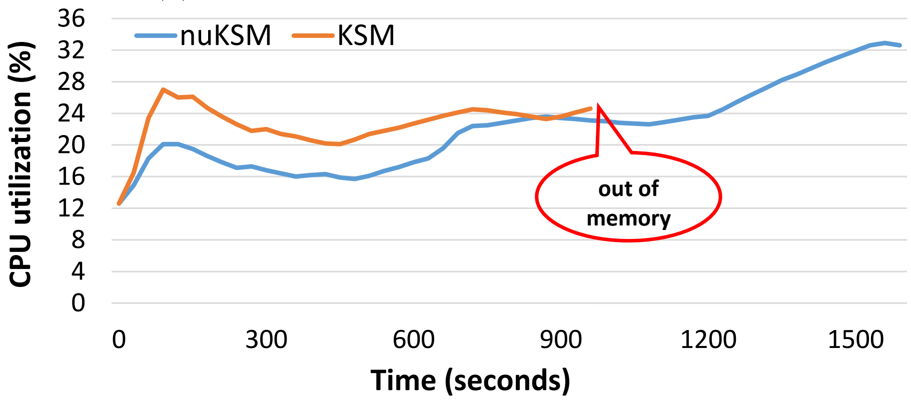
Figure 5.6 shows the results of the above experiment with KSM and nuKSM. Figure 5.6a shows that KSM throws an Out-of-Memory (OOM) error at about 900 seconds. This hap-



(a) Amount of free memory over time



(b) Amount of de-duplicated memory over time



(c) Average CPU utilization of KSM

Figure 5.6: Amount of free and de-duplicated memory, and average CPU utilization with KSM and nuKSM. KSM runs out of memory at about 900 seconds due to increasing memory pressure. nuKSM runs to completion due to faster de-duplication.

pens when the background job makes an allocation request but free memory is not available. Figure 5.6b shows the amount of memory de-duplicated over time which confirms that KSM was not able to de-duplicate enough memory before OOM occurred. Recall from Table 5.2 that KSM de-duplicated about 11 GiB memory for XSBench in our experiments in Section 5.2. However, in that case, only 20 GiB memory was in use, while a total of 180 GiB memory is in use here. Since KSM's larger trees due to larger memory size increases time to find pages with identical contents, the rate of de-duplication is low here. Thus, the amount of memory de-duplicated by KSM is hardly noticeable before the OOM in Figure 5.6b. Repeated runs of the same experiment show that if the kernel kills the background job due to OOM, instead of XSbench, then memory from XSBench's two instances starts being de-duplicated from around 1000 seconds. But that is too late to prevent OOM.

For the same experiment, nuKSM with 1800 trees is able to run XSBench instances and the background workload to completion, due to faster de-duplication. Figure 5.6b confirms that nuKSM was able to de-deuplicate more than 6 GiB memory within 900 seconds and about 40 GiB overall. Better responsiveness of nuKSM, therefore, prevented the OOM. Figure 5.6c shows the average CPU utilization of the de-deduplication thread. It also shows that nuKSM de-duplicates memory more efficiently than KSM since it is able to de-duplicate memory faster with slightly lower CPU utilization than KSM.

5.5 Comparison with UKSM

We are unaware of any published work on the effect of NUMA on de-duplication. However, to quantitatively compare against related work, we experimented with UKSM[40]. That work prioritizes memory regions for faster de-duplication based on the observation that spatially co-located regions exhibit similar de-duplication behavior. Unfortunately, though, UKSM fails to properly deduplicate pages across virtual machines (KVM). Specifically, it continuously de-merges (duplicates) pages immediately after de-duplication, even on read accesses, and thus, provides no memory savings. We reported this issue to the authors but could not be fixed so far. This forced our experiments to be limited to the bare metal system only. On bare-metal system however, applications need to be modified to use madvise system call for registering memory for de-duplication with KSM (Section 2.1).

We conducted an experiment similar to the one discussed in Section 3.1 where two identical applications were placed on different sockets (nodes). We ran two identical instances of RandomAccess workload on a bare-metal system, each on a different NUMA node. We did not run all workloads since each needs to be modified to register their memory for de-duplication and is unnecessary for this experiment's primary purpose.

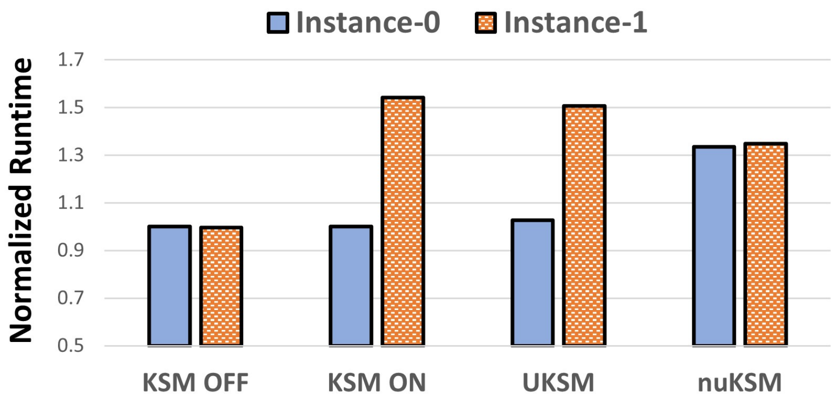


Figure 5.7: Execution time of two instances of RandomAccess executing on separate NUMA nodes with KSM (turned off and turned on), nuKSM and UKSM. Execution time is normalized to the runtime of Instance-0 with KSM OFF.

Figure 5.7 shows the result of the experiments. Like KSM, UKSM also introduces large performance variability among applications running on different NUMA nodes. Similar to KSM, we observed a performance difference of 50% in UKSM. In contrast, the difference is negligible in nuKSM. In short, we quantitatively demonstrate that state-of-art academic proposals on de-duplication suffer from the same NUMA-unawareness as Linux's KSM.

Chapter 6

Related work

Techniques to achieve high memory consolidation have been extensively studied in the literature [7, 8, 9, 18, 22, 25, 26, 27, 32, 35, 39, 40]. We discuss some important related works below, and discuss how nuKSM is different from these systems.

6.1 Content-based memory de-duplication

The seminal work in VMware ESX server pioneered content-based memory de-duplication for virtualized environments [39]. In [39], randomly selected pages are first hashed to check for similarity, and if their hashes are identical, full-page comparison is used to ensure that they can be safely de-duplicated using a copy-on-write mapping. Many memory de-duplication techniques draw inspiration from this work. Active memory de-duplication in IBM Power systems uses a similar approach to improve memory consolidation [34]. Xen hypervisor also adopted a similar approach [8], but the authors used a more efficient hashing scheme. Only two 64-byte blocks at fixed locations from the pages are hashed for similarity checks. In contrast, KSM in the Linux kernel uses full-page comparisons to perform similarity check, instead of two-step hashing and full-page comparison. In our work, we use the same approach for similarity checking as used in Linux but avoid unnecessary page comparisons using a collection of trees. More importantly, nuKSM is the first to bring attention to NUMA implications of de-duplication.

6.2 Optimizations to reduce de-duplication overheads

Over the years, several researchers have proposed various optimizations to achieve higher memory savings at less overheads. Difference Engine [22] employs a combination of sub-page level sharing and in-core memory compression to achieve high memory consolidation. Sub-page level sharing eliminates redundant content at a finer granularity than a page. Singleton [35] extends

KSM to eliminate redundancy due to multiple disk caches in a virtual environment. Catalyst uses a hashing based page comparison but offloads hash computation to a GPU for fast de-duplication [19].

6.3 Classification based memory de-duplication

CMD [8] is a classification based de-duplication approach in which pages are classified based on access characteristics. It divides a page into eight sub-pages, each with a dirty bit to indicate whether it is modified between two scans. CMD uses different stable and unstable trees for each class to avoid unnecessary page comparisons. However, CMD requires dedicated hardware support to monitor system I/O hints which introduces deployment complexities. Similarly, SmartKSM [7] classifies pages into five groups based on the type *i.e.,* free, kernel, anonymous, page cache, and inodes. nuKSM's approach of using many trees resembles that of CMD and SmartKSM but one where classification is based on the content of a page, and not the type or page access characteristics. Note that the height of the trees in CMD and SmartKSM still remains unbounded as there are only a handful of page classes. In contrast, nuKSM bounds the height by adjusting the number of trees based on the size of memory.

6.4 Fine-grained tuning of KSM parameters

Adaptive approaches for fine-grained balancing between memory sharing and de-duplication overhead have also been proposed. For example, ksmtuned [15] adjusts the scan rate of KSM based on the state of memory at runtime. When free memory falls below a certain (configurable) threshold, ksmtuned increases the scan rate to reduce memory pressure. The scan rate is reduced when free memory reaches above the specified threshold to save CPU cycles. This approach is orthogonal to nuKSM and can work alongside it. UKSM [40] prioritizes different memory regions to accelerate de-duplication, based on the observation that spatially co-located regions exhibit similar de-duplication patterns.

6.5 Hinting the memory de-duplication subsystem

KSM++ [25] and XLH [26] utilize I/O hints from the host virtual file system layer (VFS) for early detection of merging opportunities whenever VMs access their backing store to load similar libraries, configuration files, or data from their virtual disk images. In this approach, potential de-duplication candidates identified via I/O hints are prioritized for scanning to improve the de-duplication system's responsiveness.

The use of paravirtualization has been explored in different works to bridge the semantic gap between the guest OS and hypervisor. Satori [27] is a paravirtualization based approach that de-duplicates guest's file-backed pages with sharing-aware virtual block devices in Xen. A similar approach [5] was used to selectively merge anonymous (e.g., stack and heap) memory pages or free pages of different virtual machines. While being useful, the use of paravirtualization makes it harder to adopt widely.

6.6 Conflicts in memory subsystem

Besides de-duplication, other conflicts in the memory subsystem have also been discovered [12, 20, 21, 23, 28, 30]. For example, large pages improve performance by reducing the number of TLB misses. However, use of large pages could preclude memory consolidation due to reduced de-duplication opportunities [21, 23, 30] and internal fragmentation [23, 28]. While large page improve address translation performance in general, they can increase NUMA-Tax due to coarse-grained data placement [12, 20]. In contrast, we highlight the conflict between the goals of memory consolidation and NUMA locality optimizations on multi-socket servers.

6.7 Security implications of de-duplication

Since de-duplicated pages are marked Copy-on-Write, write to a de-duplicated page causes page fault. Consequently, write to a de-duplicated page is significantly slower than to a page that is not de-duplicated. Previous works have shown that this differential in access latency can potentially be exploited to leak information among co-locating VMs [24, 36, 41]. These attacks are used to leak information such as version of software running on a co-located VM [24] or for deciphering existence of a specific application in a co-located VM [36]. However, no known attack leaking data using the aforementioned channel exists.

Several countermeasures for such a channel has been proposed too. Jens et. al.[24] proposed a technique to deceive attackers by placing the binaries of specific versions of the applications that are not running on the VMs. Suzaki et al.[36] discussed that making the victim OS use obfuscation code to change runtime memory image can prevent the attack. In our work, we do not focus on the security aspects of KSM and instead assume that previously proposed defenses can be employed if side channel is a concern.

6.8 Our work

Different from these systems, *our main contribution is identification of* NUMA *implications of memory de-duplication on multi-socket servers and proposing ways to mitigate its ill-effects.*. Therefore, nuKSM is orthogonal to these prior works. Many of these existing solutions can be integrated with nuKSM to further improve its performance. For example, specialized accelerators can be used for faster checksum computation, and the scan rate of nuKSM can be adjusted at runtime. Sub-page sharing, compression, and I/O hints based de-duplication are all compatible with our design of nuKSM.

Chapter 7

Conclusion

We demonstrate that memory de-duplication can have unintended consequence to NUMA overheads experienced by applications running on multi-socket servers. Linux's memory de-duplication subsystem, namely KSM, is NUMA unaware. Consequently, while de-duplicating pages across NUMA nodes, it can place de-duplicated pages in a manner that can lead to significant performance variations, unfairness and subvert process priority.

We introduce NUMA-aware KSM, a.k.a., nuKSM, that makes judicious decisions about the placement of de-duplicated pages to reduce impact of NUMA and unfairness in execution. nuKSM also enables user to specify if it should strive to avoid placing NUMA-Tax based on the priority of processes, instead of trying to be fair to all applications/VMs. In short, it enables user a control over how NUMA-Tax due to KSM should be distributed. Finally, independent of the NUMA effect, we observed KSM fails to scale well to large memory systems due to its centralized design. We thus extended nuKSM to adopt a de-centralized design to scale to larger memory.

Bibliography

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 283–300, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378468. URL https://doi.org/10.1145/3373376.3378468. 1, 2, 7

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Proceedings of the Linux symposium*, pages 19–28. Citeseer, 2009. 1

[3] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The nas parallel benchmarks. *Int. J. High Perform. Comput. Appl.*, 5(3):63–73, September 1991. ISSN 1094-3420. doi: 10.1177/109434209100500306. URL https://doi.org/10.1177/109434209100500306. 24

[4] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for numa memory management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, page 19–31, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913388. doi: 10.1145/74850.74854. URL https://doi.org/10.1145/74850.74854. 1, 7

[5] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 143–156, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919165. doi: 10.1145/268998.266672. URL https://doi.org/10.1145/268998.266672. 37

[6] Richard W. Carr. *Virtual Memory Management*. University of Michigan Press, USA, 1984. ISBN 0835715337. 18

[7] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. Smartksm: A vmm-based memory deduplication scanner for virtual machines. *SOSP Poster*, 2013. 35, 36

[8] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. Cmd: Classification-based memory deduplication through page access characteristics. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, page 65–76, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327640. doi: 10.1145/2576195.2576204. URL https://doi.org/10.1145/2576195.2576204. 1, 35, 36

[9] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory deduplication and migration. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, page 51–62, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450312660. doi: 10.1145/2451512.2451525. URL https://doi.org/10.1145/2451512.2451525. 1, 35

[10] MySQL Community. Mysql benchmark tool. Online https://dev.mysql.com/downloads/benchmarks.html. 24

[11] Jonathan Corbet. Autonuma: the other approach to numa scheduling. Online https://lwn.net/Articles/488709/, . 8

[12] Jonathan Corbet. Transparent huge pages, numa locality, and performance regressions. Online https://lwn.net/Articles/787434/, . 1, 37

[13] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 381–394, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318709. doi: 10.1145/2451116.2451157. URL https://doi.org/10.1145/2451116.2451157. 1, 2, 7

[14] Linux Kernel Documentation. Kernel samepage merging. Online https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html, . 1, 5

[15] Red Hat Documentation. The ksm tuning service. Online https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_tuning_and_optimization_guide/sect-ksm-the_ksm_tuning_service, . 36

[16] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, May 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.44. URL https://doi.org/10.1109/MM.2008.44. 25

[17] A NUMA API for LINUX. Technical linux whitepaper. Online http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf. 8

[18] Y. Fu, H. Jiang, N. Xiao, L. Tian, and F. Liu. Aa-dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In *2011 IEEE International Conference on Cluster Computing*, pages 112–120, 2011. doi: 10.1109/CLUSTER.2011.20. 35

[19] Anshuj Garg, Debadatta Mishra, and Purushottam Kulkarni. Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, page 44–59, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349482. doi: 10.1145/3050748.3050760. URL https://doi.org/10.1145/3050748.3050760. 36

[20] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. Large pages may be harmful on NUMA systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 231–242, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/gaud. 37

[21] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. Smartmd: A high performance deduplication engine with mixed pages. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 733–744. USENIX Association, 2017. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/guo-fan. 37

[22] 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. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 309–322, USA, 2008. USENIX Association. 1, 35

[23] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 705–721, USA, 2016. USENIX Association. ISBN 9781931971331. 1, 18, 37

[24] Jens Lindemann and Mathias Fischer. A memory-deduplication side-channel attack to detect applications in co-resident virtual machines. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 183–192, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450351911. doi: 10.1145/3167132.3167151. URL `https://doi.org/10.1145/3167132.3167151`. 37

[25] 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 *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12), London, UK, March 3, 2012*, 2012. 35, 36

[26] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 279–290, San Jose, CA, June 2013. USENIX Association. ISBN 978-1-931971-01-0. URL `https://www.usenix.org/conference/atc13/technical-sessions/presentation/miller`. 35, 36

[27] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 1, USA, 2009. USENIX Association. 35, 36

[28] Ashish Panwar, Sorav Bansal, and K. Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 347–360, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304064. URL `https://doi.org/10.1145/3297858.3304064`. 1, 18, 37

[29] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Avoiding scheduler subversion using scheduler-cooperative locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387521. URL https://doi.org/10.1145/3342195.3387521. 15

[30] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 1–12, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340342. doi: 10.1145/2830772.2830773. URL https://doi.org/10.1145/2830772.2830773. 37

[31] Mitosis Project. Btree. Online https://github.com/mitosis-project/mitosis-workload-btree. 24

[32] S. Rachamalla, D. Mishra, and P. Kulkarni. Share-o-meter: An empirical analysis of ksm based memory sharing in virtualized systems. In *20th Annual International Conference on High Performance Computing*, pages 59–68, 2013. doi: 10.1109/HiPC.2013.6799096. 35

[33] Rik van Riel. Page replacement in linux 2.4 memory management. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, page 165–172, USA, 2001. USENIX Association. ISBN 1880446103. 18

[34] Breno Leitao Rodrigo Ceron, Rafael Folco and Humberto Tsubamoto. Power systems memory deduplication. Online http://www.redbooks.ibm.com/redpapers/pdfs/redp4827.pdf, 2017. 5, 35

[35] Prateek Sharma and Purushottam 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, page 15–26, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450308052. doi: 10.1145/2287076.2287081. URL https://doi.org/10.1145/2287076.2287081. 1, 35

[36] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest os. In *Proceedings of the Fourth European Workshop on*

System Security, EUROSEC '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306133. doi: 10.1145/1972551.1972552. URL https://doi.org/10.1145/1972551.1972552. 37

[37] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. URL https://www.mcs.anl.gov/papers/P5064-0114.pdf. 24

[38] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on cc-numa compute servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 279–289, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917677. doi: 10.1145/237090.237205. URL https://doi.org/10.1145/237090.237205. 1, 7

[39] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2003. ISSN 0163-5980. doi: 10.1145/844128.844146. URL https://doi.org/10.1145/844128.844146. 1, 5, 35

[40] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. UKSM: swift memory deduplication via hierarchical and adaptive memory region distilling. In Nitin Agrawal and Raju Rangaswami, editors, *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, pages 325–340. USENIX Association, 2018. URL https://www.usenix.org/conference/fast18/presentation/xia. 1, 33, 35, 36

[41] J. Xiao, Z. Xu, H. Huang, and H. Wang. Security implications of memory deduplication in a virtualized environment. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013. doi: 10.1109/DSN.2013.6575349. 37

[42] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 89–102, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605584829. doi: 10.1145/1519065.1519076. URL https://doi.org/10.1145/1519065.1519076. 18