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

Akash Panda*, Ashish Panwar, Arkaprava Basu

Department of Computer Science and Automation
Indian Institute of Science
{*akashpanda, ashishpanwar, arkapravab*}*@iisc.ac.in*

***Abstract*—An operating system has many memory management goals including reducing memory access latency, and 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 (non-uniform memory access) 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. Toward this, 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, unfairness, and avoid priority subversion. Independent of the NUMA effects, we observed that KSM scales poorly to systems with larger memory sizes due to its centralized design. Thus, we extended nuKSM to adopt a de-centralized design.

I. Introduction

Memory management is a crucial piece in the design of a computing system. It has several responsibilities ranging from ensuring quick access to data to enabling memory consolidation. For example, the placement of pages in multi-socket NUMA (non-uniform memory access) servers impacts memory access latency that an application experiences [1]–[4]. Similarly, memory de-duplication plays a role in memory consolidation and over-commitment [5]–[9].

Different memory management goals are known to conflict with each other [10]–[12]. This happens when independent subsystems are responsible for different goals and each works in its own silo [12]. In this work, we describe a previously unreported conflict – how Linux's de-duplication efforts may conflict with its NUMA management.

De-duplication plays an important role in memory consolidation and over-commitment, particularly under virtualization [7], [9], [13]. It is not uncommon for different virtual machines (VMs) to run the same or similar OSes, libraries, and applications [14]. Several services are often replicated for better load balancing and fault tolerance [7]. Consequently, many physical pages belonging to different VMs can have the same content. When such VM instances run on the same machine, it provides opportunities to consolidate the memory usage through de-duplication. Linux's Kernel Same page Merging (KSM [15]), VMware's Transparent Page Sharing (TPS [8]) are real-world examples of subsystems responsible for de-duplication.

Linux's KSM periodically scans the contents of pages mapped to different process's virtual address spaces[1], including those mapped by virtual machine's (here, KVM) guest physical address space. When it finds two pages with same contents, it de-duplicates them. One of the copies is retained while the other is freed. Mappings to the retained copy (de-duplicated page) from the sharing processes are rendered copy-on-write. If a process later attempts to write to a de-duplicated page, KSM first creates a new copy of that page.

To enable greater de-duplication opportunities, KSM enables de-duplication of pages across different sockets (NUMA nodes) in multi-socket servers. In such servers, memory access latencies can be non-uniform. The portion of the physical memory that reside on the same socket as the processor accessing the memory is local to that processor. The memory on other sockets are remote to a given processor. Remote memory accesses are typically 1.5–2× slower than the local accesses [3], [4]. Thus, most of the memory accesses should be local for good performance.

De-duplication across NUMA nodes can induce remote memory accesses in one or more of the VMs whose pages are de-duplicated. When pages on different NUMA nodes with identical contents are de-duplicated, KSM retains one of the copies on one of the nodes. Consequently, subsequent accesses to the de-duplicated copy of the page from a VM running on a different node become remote.

Unfortunately, we discovered that KSM is unaware of the NUMA implications of 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, as well as unfairness in execution. We demonstrate that a VM could experience significantly more remote accesses (*e.g.,* more

* Author is currently affiliated with AMD. This work was performed when the author was a student at Indian Institute of Science.

[1]We use VMs and processes interchangeably since in the Linux/KVM ecosystem, VMs are KVM processes to Linux (hypervisor).

978-1-6654-4278-7/21/$31.00 ©2021 IEEE
DOI 10.1109/PACT52795.2021.00026

than 90%) than another, even when both execute instances of the same application. Consequently, the performance of two identical virtual machines can differ by as much as 46%.

After de-duplication of pages across nodes, a VM's performance hinges on whether its copy was retained or was 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 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 order in which the processes/VMs were created and, thus, is arbitrary.

KSM is also oblivious to process priorities. When coupled with NUMA-unawareness, this can lead to priority subversion where a higher priority process slows down more than a lower priority one due to NUMA effects. Users have no control over which processes or virtual machines should enjoy more local accesses to de-duplicated pages. As a consequence, there is no way in Linux to control the performance implications of NUMA while using system-wide de-duplication for better memory consolidation and over-commitment.

To address these, we introduce nuKSM– NUMA-aware KSM. The foremost objective of nuKSM is to make an *informed* choice about which of copy of the pages with duplicate contents to retain and which one to free. We strive to reduce the total number of remote accesses by keeping a de-duplicated page close to the VM that is accessing it more frequently. The frequency of accesses to the remote memory decides the NUMA overheads. To determine which copy is more frequently accessed, nuKSM leverages information on page access frequency already available in Linux for page reclamation (*e.g., active* and *inactive* lists).

It is however possible, that both VMs frequently access their copies of the page being de-duplicated. There, nuKSM strives to spread the NUMA overheads equitably among the VMs by distributing de-duplicated pages across nodes in a round-robin fashion. This improves fairness and avoids performance variations. We empirically find that nuKSM reduces performance variations in certain workloads from 46% to a mere 4%.

To avoid priority subversion, nuKSM enables the user to request enforcement of process priorities in the distribution of NUMA overheads due to de-duplication. When requested, 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 will find most of its de-duplicated pages on its local node.

Beyond NUMA unawareness, we notice that KSM scales poorly with the size of the memory. We find that KSM uses centralized data structures, like a single red-black tree to track *all* de-duplicated pages and one more for *all* de-duplication candidate pages. The latency to access and update these structures increases with memory size. Therefore, the responsiveness of KSM (time taken to remove duplicates) degrades on systems with large memory sizes.

Instead of two centralized trees, nuKSM keeps two forests. A system's memory capacity determines the number of trees in the forests. For functional correctness, it is imperative for all duplicate copies be assigned to the same tree. Therefore, nuKSM decides the tree that will track a page based on the checksum of its contents. Pages with the same content are bound to have the same checksum. Thus, 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 can lead to unfairness, performance variability, and priority subversions among the concurrently running VMs.
- We created nuKSM, NUMA-aware memory de-duplication in Linux that makes judicious decision on the placement of the de-duplicated pages to reduce NUMA overheads, ensure fairness or to avoid priority subversion.
- We further observed that KSM scale poorly with increasing memory sizes. We address this by de-centralizing the process of finding de-duplication candidates in nuKSM.

II. Background

A. Memory de-duplication in Linux

It is not uncommon for pages to have duplicate contents, especially when multiple VMs are hosted on a single server. VMs 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 usage. While many OSes and hypervisors support de-duplication [8], [15], [16], we focus on KSM in the Linux and KVM ecosystem. A process needs to register its virtual memory regions with KSM using the madvise system call with the MADV_MERGEABLE flag for it to be considered for de-duplication [15]. KVM, however, automatically registers the entire memory of VMs to KSM.

Figure 1 shows an overview of the data structures used in KSM. The algorithm to detect pages with duplicate content is an important but resource-consuming building block of memory de-duplication. KSM maintains two red-black trees for the purpose, namely stable and unstable. The tree nodes are arranged based on page contents. Already de-duplicated pages are placed in the stable tree, while all potential candidates are placed in the unstable tree. A page whose content has not been updated between the two most recent scans is considered a de-duplication candidate. Frequently updated pages are unlikely to be de-duplicated with others, and thus, ignored by KSM. Further, the cost of identifying and de-duplicating a page is amortized only if it stays de-duplicated for a long duration. A write to a de-duplicated page incurs a page (COW) fault and copying of the contents.

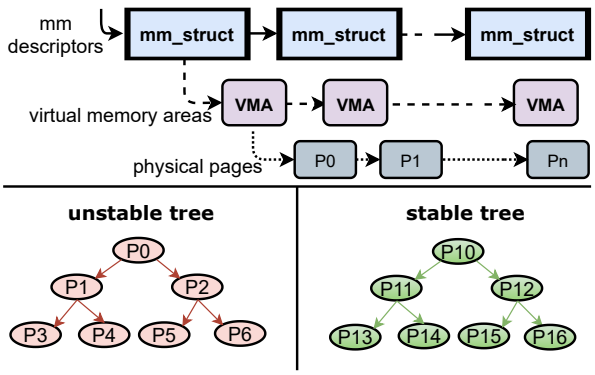


Figure 1: Key data structures in KSM. A mm_struct represents a virtual address space. KSM chooses address spaces to scan from the list of registered mm_structs. A VMA represents a contiguous address region. KSM uses unstable and stable trees for identifying de-duplication candidates.

KSM employs a kernel thread for ① finding de-duplication opportunities and ② for performing the de-duplication. Processes, including VMs, that are registered with KSM are kept in a list (top of Figure 1). The kernel thread periodically scans processes from the list, one at a time. During a scan, KSM sequentially runs through the physical pages mapped to process's virtual memory regions (struct vm_area_struct) to identify de-duplication candidates. KSM's scan rate is parameterized in Linux.

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 mapped to the freed physical page is now mapped to the de-duplicated page in the stable tree. The new mapping is rendered Copy-on-Write (COW) to prevent a process from unilaterally modifying contents of a de-duplicated page.

If there is no match in the stable tree, a checksum of the page's contents is computed. It is then compared against the page's checksum that was computed during the last scan. Checksum from the previous scan is stored in the metadata of the physical page. A mismatch between the checksums signifies that the page has been modified, and thus, it 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. On a match, the physical page frame in the unstable tree is freed. The virtual address range mapping onto the freed physical page is now mapped to the page that is being scanned. As before, the mapping is rendered COW. The de-duplicated copy is then added to the stable tree. If there was no match, the scanned page is added to the unstable tree. This page will then be compared with other candidates during the scan.

The stable tree is constructed once when KSM is initialized. However, the unstable tree is reconstructed during each scan to avoid comparisons with recently updated pages. Pages in the unstable tree are not write-protected, and their content may have been updated between scans.

B. *Non-uniform memory access (*NUMA*)*

Modern servers often sport 2-4 sockets, with one CPU in each socket [1], [2], [4]. Each socket has local memory (DRAM) 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 OS. A CPU can access both local memory and memory attached to any other sockets (remote memory). However, the latency of accessing remote memory is typically 1.5–2× higher than that of accessing local memory [3], [4]. This gives rise to non-uniform memory access or NUMA.

A goal of OS's memory management is to reduce overheads due to remote memory accesses. For simplicity, we will refer to the overhead of accessing remote memory as *NUMA-Tax*. Linux provides tools like numactl and libnuma library that allows users/programmers to explicitly bind memory to specific NUMA nodes and migrate pages from one node to other [17]. Linux also supports automatic page migration via AutoNUMA [18]. 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 at regular intervals to identify local and remote accesses. AutoNUMA migrates pages that are accessed frequently from a remote CPU and avoids migrating any pages that are accessed from different CPUs. Following the same principle, AutoNUMA does not to migrate de-duplicated pages.

III. 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, 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 KSM leads to avoidable and unequal distribution of NUMA-Tax across 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 servers with many sockets. In an ideal world, one would thus allow de-duplication across nodes but expect

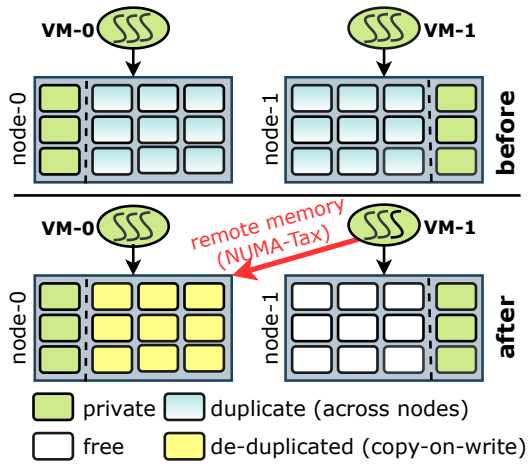
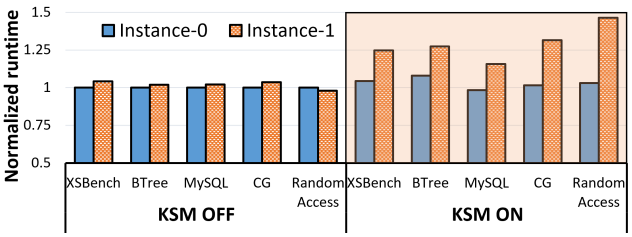
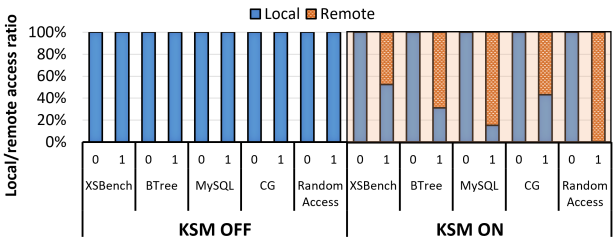


Figure 2: Effect of de-duplication on NUMA-Tax. Both VMs access local memory prior to de-duplication (top). After de-duplication (bottom), all merged pages are placed on node-0.



(a) Normalized execution time



(b) Local/remote memory access ratio

Figure 3: 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 those of Instance-0 s without KSM.

memory management to contain the ill-effects of NUMA-Tax. In this work, we make progress toward this goal. We start by quantifying the ill-effects of NUMA-Tax induced by de-duplication in Linux/KVM.

A. 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 to the fact that KSM chooses where to place a de-duplicated physical page based on the order in which it scans processes, oblivious of the NUMA considerations. As discussed in § II-A, when the content of a page being scanned is identical to that of an existing candidate page in the unstable tree, KSM always retains the page being scanned and frees the one in the unstable tree. When the page being scanned matches a page in the stable tree, the one from the tree is retained. In either case, which page to retain is decided by the order in which KSM happens to scan address spaces, disregarding NUMA implications. Also, 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 on one or a subset of the VMs running on different nodes.

We demonstrate this behavior with a simple example. Figure 2 shows a two-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 before 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, pages of VM-1 are scanned and 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 VM-0's pages match those of VM-1 in the unstable tree, the pages of VM-0 residing on node-0 are retained while VM-1's copies on node-1 are freed. This way, all de-duplicated pages get placed on node-0. Consequently, VM-1 experiences high and an unfair share of NUMA-Tax after de-duplication, while VM-0 continues to enjoys local accesses.

We quantify the ill-effects of KSM on NUMA-Tax for a set of applications on a two-socket server (§ V-A details methodology). We set up two virtual machines VM-0 and VM-1, each executing an instance of the same application concurrently. 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. Figure 3a reports the performance of each application. The performance of each instance is normalized to the runtime of Instance-0 of the given application *without* KSM.

When KSM is disabled, performances of both instances are similar. Since instances run on different nodes, there is little CPU and memory contention. However, when KSM is enabled, we observe a significant performance difference (16%-46%) among the instances of the same application. For example, RandomAccess and CG in VM-1 slow down by 46% and 31% compared to when KSM was disabled while they are 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 get added first to the unstable tree. Therefore,

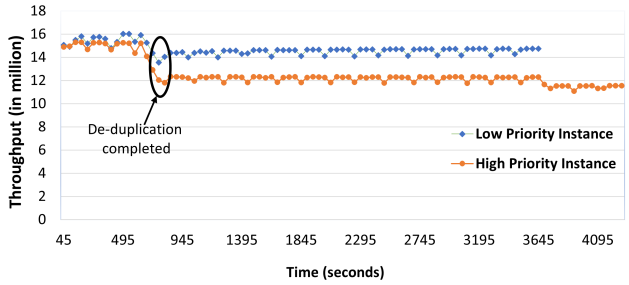


Figure 4: Throughput of two identical instances of BTree running with different priorities with KSM.

VM-0's pages are retained on node-0 after de-duplication. Figure 3b shows the breakdown of application's DRAM accesses (*i.e.,* after missing in the cache) 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 3a is a direct consequence of larger fractions of slow remote accesses experienced by Instance-1.

B. Priority subversion

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

KSM's NUMA-unawareness makes it vulnerable to priority subversion. Priority subversion is the circumstance where a higher priority process gets penalized by a low priority process due to priority-unaware resource allocation [19].

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 4 shows the effect of de-duplication by KSM on the throughput. 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. 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.

C. Low responsiveness with large memory

Observation: *KSM scales poorly with larger memory sizes.*

Beyond NUMA, we noticed that KSM scales poorly to large memory systems. This is due to the centralized nature of KSM's data structures. It maintains one set of trees for the entire memory. As memory size grows, the height of the trees grow. While scanning, KSM compares each page with stable tree nodes and then (if needed) with unstable tree nodes to identify de-duplicate candidates. The number of comparisons grows with the height of the tree, which, in turn, grows with the size of memory. In § V-D, we show how 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.

IV. Design and Implementation

We now detail the design of nuKSM. Our design objectives include ① minimizing performance variability and unfairness due to memory de-duplication across NUMA nodes, ② the ability to distribute NUMA-Tax based on the priority of different processes to avoid priority subversion, and ③, finally, improving the responsiveness of de-duplication in large memory systems. We implemented nuKSM in Linux kernel version 5.4.0 by extending KSM.

A. 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 if one of its infrequently accessed pages is de-duplicated and is placed 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 avoids paying NUMA-Tax when possible.

We provide a simple example to illustrate this policy in practice. 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. Also assume that P2 and P3 are accessed by both VMs with similar frequency, and P4 is inactive. In this example, nuKSM would place P0 close to VM-0, P1 close to VM-1. For an even distribution of NUMA-Tax, it would place one of P2 & P3 close to one VM and the other close to the other. Since P4 is inactive, its placement has little impact on performance. nuKSM could place P4 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 this policy requires knowing the 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 the corresponding pages. The OS can periodically clear access bits and check

them after a while to find which pages are being accessed frequently [10], [11], [20]. 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 overhead of traversing the page tables [11], [20].

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 [21]. It 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 are kept in the active list. Under memory pressure, pages from the inactive list are swapped to the storage. Implementation of the page reclamation and its heuristics have been optimized over the years by the Linux community. We piggyback on the hints about a page's access frequency already available from the page reclamation system to realize nuKSM's policy of which copy of pages with identical content to retain.

As an example, assume nuKSM is to de-duplicate two pages with same contents from two nodes– say P0 from node-0 and P1 from node-1. nuKSM first checks 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 de-duplicated page and free the other copy (P1). If both P0 and P1 are in active list, nuKSM uses a round-robin policy to determine which copy to free. For example, if the first of two active pages are placed on node-0 while de-duplicating, the next one would be placed on node-1, and so on. This ensures that the frequently accessed pages are evenly distributed across nodes. Finally, if both pages are inactive, they are also distributed evenly across nodes in round-robin. It helps in balancing memory allocation to avoid thrashing a particular node. However, there is typically no performance implications of placement of inactive pages.

B. Priority based memory de-duplication

Fairness and performance predictability are not always the most important objectives in certain execution environments. For such cases, nuKSM enables users to configure which process (VM) should enjoy more local memory accesses. If the priority-based de-duplication is enabled, nuKSM distributes NUMA-Tax in the ratio of the relative priorities of the processes that share the de-duplicated pages.

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. nuKSM repurposes the nice values while de-duplicating pages under this policy. For simplicity, we add 21 to each nice value to convert it to a non-zero positive integer; 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* signifies a higher preference of making the de-duplicated page local to the process that is currently being scanned. The *nuShare* is then compared to a pseudo-random number 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 local access to the de-duplicated page from the process being scanned. 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 ensures 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.

C. Enhancing responsiveness

Independent of nuKSM's primary goal of making de-duplication NUMA-aware, it also strives to scale de-duplication better to large memory systems. As discussed earlier, KSM's low responsiveness is rooted in having one set of trees for 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 the 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 allows distributing pages across many sets of trees. Consequently, it limits 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 since 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

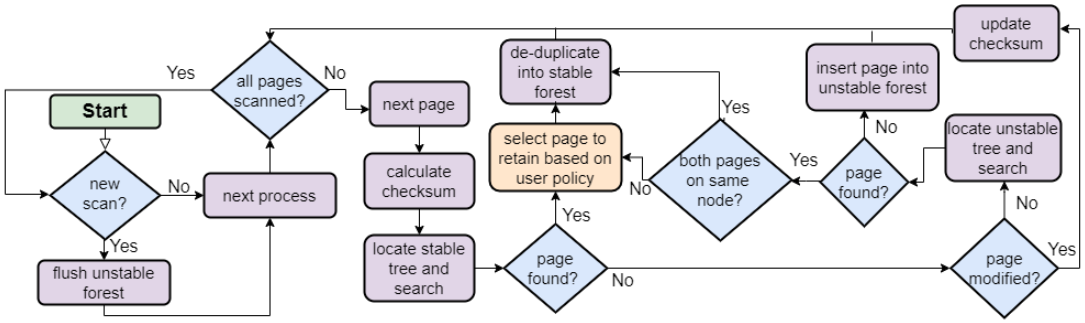


Figure 5: Memory de-duplication workflow in nuKSM.

that using one stable and one unstable trees per 100 MiB memory provides a reasonable balance between the cost and benefits of using a de-centralized forest-based approach.

D. Putting it all together

We depict the entire workflow of nuKSM with Figure 5. nuKSM starts by periodically scanning address spaces from the list of registered processes. The VMs, *i.e.,* KVM processes register their entire memory by default. The unstable trees are flushed prior to starting a new scan. A checksum of a page's contents is used to index into the forest of stable and unstable trees. The corresponding stable and unstable trees are then searched for de-duplication opportunities. On a match, the decision on which copy to retain and which one to free depends upon user settings. By default, nuKSM uses the principle described in § IV-A to ensure equitable distribution of NUMA-Tax for fairness. However, it can be changed to the one based on priority (§ IV-B), 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.

E. Scaling to many sockets

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

That leaves us to discuss how nuKSM's algorithm for fairness (§ IV-A) 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 the active lists of their respective NUMA nodes, ② only one of the pages is in the active list, and ③ both the pages are in the inactive lists. Under the first condition, *i.e.,* when both P_x and P_y are in the active list, nuKSM tries to evenly distribute the de-duplicated pages across the 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 the inactive list then there is no expected performance implications of NUMA placement. Still, nuKSM uses the same technique as used in the first condition, to evenly distribute the de-duplicated pages across NUMA nodes.

We empirically evaluated nuKSM's scalability beyond two VMs [22] but omit details here.

V. 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 does nuKSM's priority-based de-duplication help users in controlling the distribution of NUMA-Tax? and (3) how responsive is nuKSM in exploiting de-duplication opportunities in large memory systems?

A. Methodology

We conduct experiments on a dual-socket Intel Xeon Gold 6140 (Skylake) server with 18 cores and 192 GiB memory per socket. The processor runs at a base frequency of 2.30 GHz. We disable 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

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 [23]	A mini-app representing a key computation kernel of the Monte Carlo neutron transport algorithm memory footprint: 11 GiB, thread count: 4
MySQL [24]	A popular database service, benchmarked with 100 sysbench clients using in-memory tables memory footprint: 20 GiB, thread count: 1
BTree [25]	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 [26]	Implementation of congruent gradient algorithm memory footprint: 3.5 GiB, thread count: 4

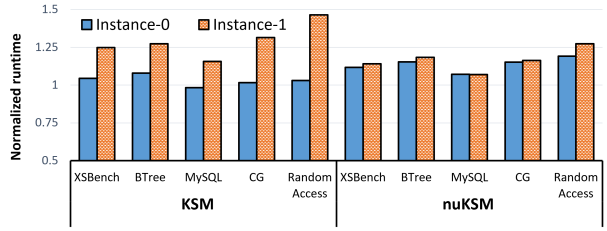
Table I: Details of the evaluation platform and benchmarks.

the host with KVM. We extend the same kernel to implement nuKSM. Both KSM and nuKSM operate at the same rate, scanning 1K pages every 100 milliseconds. Each VM is configured with four vCPUs and 30 GiB memory, unless specified otherwise. To execute a VM on a specific node, we bind its memory and vCPUs to the memory and physical CPUs of that node. 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. We evaluate with a mix of real-world databases and high-performance computing applications, and memory-intensive micro-benchmarks that are sensitive to NUMA. Table I provides further details of our evaluation platform and workloads. Appendix § A provide instructions to reproduce the results.

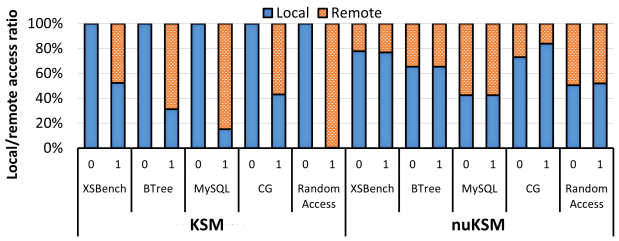
B. Memory de-duplication for fairness

We first evaluate how nuKSM's NUMA-awareness helps in moderating arbitrary performance variability and in ensuring fairness among co-running VMs. We conduct an experiment similar to the one discussed in § III-A. Two VMs running identical applications were placed on different nodes. As seen in Figure 6a, KSM introduces high performance variability and thus, unfairness amongst applications running on different VMs, ranging from 15% performance difference between two instances of MySQL to 46% for those of RandomAccess. In contrast, the difference is negligible with nuKSM. At most 4% variability was observed for RandomAccess.

Figure 6b shows the percentages of local and remote memory access for both instances of each application, with KSM and nuKSM, respectively. As discussed in § III-A, different instances of an application witness different amounts of remote memory accesses under KSM. However, with nuKSM, the remote access percentages are almost equal across both the instances for all applications. This confirms that nuKSM distributes NUMA-Tax fairly that helps to avoid variability and unfairness in application performances.



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



(b) Local/remote memory access ratio

Figure 6: Performance variability and local/remote access ratios of two identical application instances (0 and 1) with KSM and nuKSM.

We quantify fairness (or lack thereof), using a well-known metric that is used to measure performance in multi-programmed workloads [27]. For two instances of an application *I0* and *I1*, fairness is calculated as follows:

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

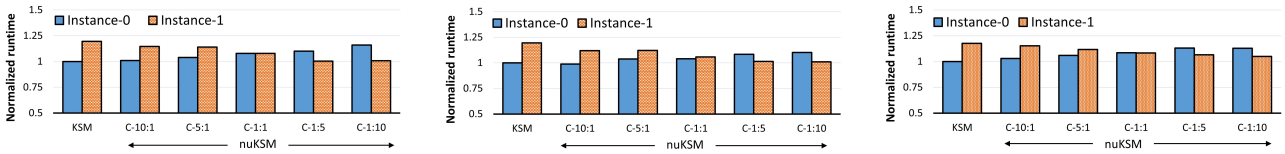
The *slowdown* is measured with respect to the baseline system. In our case, the baseline represents the case where de-deduplication (KSM) is disabled. Note that the value of fairness lies between 0 and 1. A higher value of fairness is desirable as it signifies low-performance variation.

Table II 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

Benchmark	fairness		perf. of	memory saved (GiB)	
	KSM	nuKSM	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 II: Amount of de-duplicated memory and fairness with KSM and nuKSM. Rightmost column shows the combined performance of nuKSM, normalized to KSM.



(a) BTree (b) XSBench (c) MySQL

Figure 7: Execution time of two instances of different applications executing on different nodes with different priorities in KSM and nuKSM. Execution time is normalized to the runtime of Instance-0 with KSM.

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

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 the instances of an application. For normalization, the combined runtime in nuKSM is then divided by that under KSM. Normalization helps discard instance-specific runtime differences and provides a measure of overall system throughput. Table II 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, in the last set of sub-columns of Table II, we report memory saved by KSM and nuKSM. Clearly, nuKSM is at least as effective as KSM in saving memory, while also ensuring fairness.

C. Priority based memory de-duplication

In § III-B, we demonstrated how KSM subverts priority goals with users having no control over the distribution of NUMA-Tax. 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 III. The table also shows the fraction of de-duplicated pages that are local to each VM after nuKSM completed de-duplication. 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 7 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 to process priorities. Hence, KSM is shown once for this experiment.

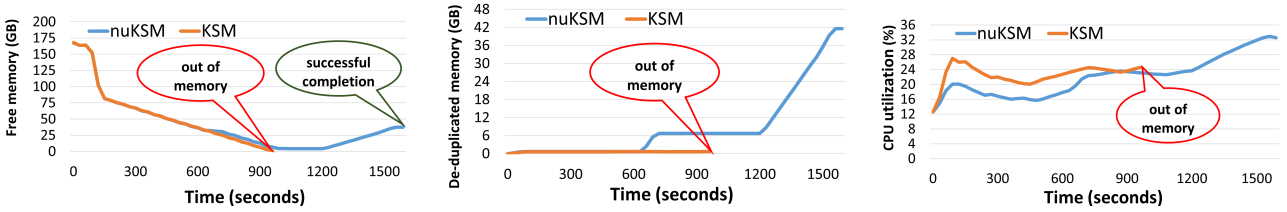
The relative priority of VM-0 decreases from left to right in each sub-figure of Figure 7. Consequently, the fraction of de-duplicated pages that is local to Instance-0 also decreases from left to right *i.e.,* from 91% in C-10:1 to 50% in C-1:1, and further to only 9% in C-1:10. At the same time, the 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.

Overall, Figure 7 shows that nuKSM can distribute NUMA-Tax accurately and at a fine grain based on the relative priorities assigned by the user. Note that both instances perform similarly in C-1:1. This configuration represents a special case wherein both VMs run with the same priority, and hence nuKSM ensures fairness.

D. Responsiveness in large memory systems

We now demonstrate how nuKSM's de-centralized design improves 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 of physical memory every 15 seconds. The background job allocates a total of 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. A node has about 180 GiB memory available. The combined memory footprint of all three processes is slightly higher than the available memory. Hence, the system will run out of memory if de-duplication does not free memory fast enough, *i.e.,* if not responsive enough. To cater to the larger memory size,



(a) Free memory over time (b) De-duplicated memory over time (c) Average CPU utilization of KSM

Figure 8: 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.

we also configure the scan rate to 10K pages every 100 milliseconds in both KSM and nuKSM.

Figure 8 shows the results of this experiment with KSM and nuKSM. Figure 8a shows that KSM throws an Out-of-Memory (OOM) error at about 900 seconds. This happens when the background job makes an allocation request but free memory is unavailable. Figure 8b shows the amount of memory de-duplicated over time which confirms that KSM was unable to de-duplicate enough memory before OOM occurred. Recall from Table II that KSM de-duplicated about 11 GiB of memory for XSBench in our experiments in § V-B. However, in that case, only 20 GiB of memory was in use, while a total of 180 GiB of memory is in use here. KSM's larger trees due to larger memory size increase the time to find pages with identical content. Consequently, de-duplication slows down. Thus, the amount of memory de-duplicated by KSM is hardly noticeable before the OOM in Figure 8b. 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 is able to run XSBench instances and the background workload to completion, due to faster de-duplication. Figure 8b 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 8c 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.

E. *Comparison with a related work:* UKSM

We are unaware of any published work on the implications of NUMA on de-duplication. However, to quantitatively compare against a related academic work, we experimented with UKSM [6]. UKSM prioritizes memory regions for faster de-duplication based on the observation that spatially co-located regions exhibit similar de-duplication behavior. Unfortunately, UKSM fails to properly de-duplicate pages across virtual machines (KVM). Specifically, it de-merges (duplicates) pages immediately after de-duplication, even on read accesses, and thus, provides no memory savings. This forced our experiments to be limited to the bare-metal system only. On bare-metal system, applications needed to be modified to use madvise system call for registering memory for de-duplication, unlike under KVM (§ II-A).

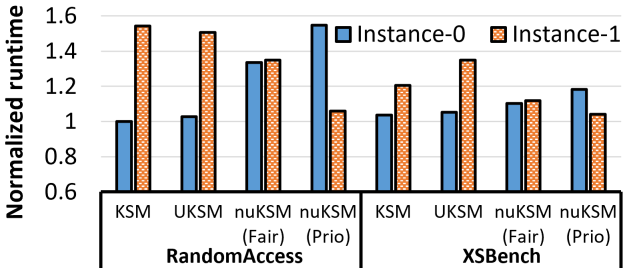


Figure 9: Execution time of two instances of applications executing on different NUMA nodes. Execution time is normalized to the runtime of Instance-0 with KSM disabled.

We conducted an experiment similar to the one discussed in § III-A. We ran two instances of a given application (here, RandomAccess and XSbench) on a bare-metal system, each on a different NUMA node. We could not run all workloads since each workload needs to be modified to register its memory for de-duplication. This becomes cumbersome for applications whose memory allocation is not straightforward, e.g., incremental memory allocation over time or the use of custom memory allocators.

Figure 9 shows the result. Like KSM, UKSM also introduces large performance variability among applications running on different nodes. We observed UKSM introduces performance variability of up to 50%. In case of XSbench the variability under UKSM is even more than that under KSM. This is because pages in XSBench are de-duplicated faster under UKSM, resulting in more pronounced NUMA effect. In short, we quantitatively demonstrate that state-of-art academic proposals suffer from the same NUMA-unawareness as Linux's KSM. In contrast, nuKSM (Fair) distributes NUMA-Tax fairly to avoid performance variability. Further, nuKSM (Prio) shows how nuKSM enables user to control the distribution of NUMA-Tax by making Instance-1 high priority and thus, reduce its runtime.

VI. Related work

Techniques for better memory consolidation have been extensively studied [6]–[9], [13], [14], [28]–[33]. VMware ESX server pioneered content-based memory de-duplication for virtualized environments [8]. It randomly selects pages to check for a match in their hashes. On a match, full-page comparison is performed before de-duplicating page contents. Active memory de-duplication in IBM Power systems uses a similar approach [16]. Xen hypervisor adopts a similar approach [14], but uses a more efficient hashing scheme. Only two 64-byte blocks at fixed locations from the pages are hashed for similarity checks. In contrast, KSM uses full-page comparisons without hashing. We extend KSM but avoid unnecessary page comparisons using a collection of comparison trees. Importantly, nuKSM is the first to report the NUMA implications of de-duplication.

Difference Engine [7] 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. Singleton [13] extends KSM to eliminate redundancy due to multiple disk caches in a virtual environment. Catalyst offloads the hash computation to GPU for quickly finding de-duplication candidates [34]. CMD [14] classifies pages based on access characteristics and divides a page into eight sub-pages, each with a dirty bit to indicate whether it was modified between two scans. CMD uses separate stable and unstable trees for each class to avoid unnecessary comparisons. However, CMD requires dedicated hardware support to monitor system I/O hints. SmartKSM [29] also classifies pages into groups based on the type *e.g.,* free, kernel, anonymous pages. nuKSM's approach of using many trees resembles that of CMD and SmartKSM. However, nuKSM chooses which tree to use based on the content of a page, and not based on the type or access characteristics. Further, nuKSM limits the height of trees by adapting the number of trees based on the size of memory while the height of the trees in CMD and SmartKSM can be arbitrarily high.

Researchers also proposed balancing memory sharing and de-duplication overheads. For example, ksmtuned [35] increases (or decreases) KSM's scan rate when free memory is below (or above) a certain threshold. This approach is orthogonal to nuKSM and can work alongside it.

KSM++ [30] and XLH [31] use I/O hints from the host file system for early detection of de-duplication opportunities when VMs access backing stores to load files or data from their virtual disks. The identified de-duplication candidates are then prioritized for scanning to quciken de-duplication. Satori [32] uses paravirtualization to de-duplicate guest's file-backed pages with sharing-aware virtual block devices in Xen. A similar approach [36] was used to selectively merge anonymous or free pages of different virtual machines. However, paravirtualization makes wider adoption harder.

Different from these, our key contributions are the identification of NUMA implications of memory de-duplication and then proposing ways to mitigate its ill-effects. Many of these existing optimizations can work with nuKSM as well. 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 compatible with nuKSM.

Researchers have reported several other conflicts among memory management goals [10]–[12], [37]–[39]. For example, large pages improve performance by reducing the number of TLB misses [40], [41]. However, use of large pages could preclude memory consolidation due to reduced de-duplication opportunities [11], [37], [38] and internal fragmentation [10], [11]. While large pages reduce address translation overheads, they can increase NUMA-Tax due to coarse-grained data placement [12], [39]. We highlight a new conflict between the memory de-duplication and NUMA management on multi-socket servers.

Previous works have shown that the difference in write latency, due to COW fault on de-duplicated pages, can possibly be exploited to leak information to co-located VMs [42]–[44]. Several countermeasures have also been proposed. Jens et al. [43] proposed deceiving the attacker by placing various non-running binaries of applications in the VM. Suzaki et al. [42] mentioned that a victim OS can prevent such attacks by changing the runtime memory image using code obfuscation We do not focus on the security aspects of KSM assuming that previously proposed defenses can be employed to overcome security concerns.

VII. Conclusion

We demonstrate that memory de-duplication can have unintended consequence to NUMA overheads in multi-socket servers. Linux's memory de-duplication subsystem, KSM, is NUMA unaware. Consequently, it can introduce significant performance variations, unfairness, and subvert process priority due to uncontrolled NUMA effects. We introduce nuKSM that makes judicious decisions on the placement of de-duplicated pages to reduce NUMA overheads and unfairness. nuKSM also enables users to control the distribution of NUMA-Tax due to de-duplication across concurrent VMs and processes. Further, nuKSM adopts a de-centralized design to scale to larger memory sizes.

VIII. Acknowledgement

This work is supported by grants from Semiconductor Research Corporation (grant number 2019-IR-2925), and from VMware Inc. Arkaprava is supported by a Young Investigator Fellowship by Pratiksha Trust, Bangalore. Ashish Panwar is supported by the Prime Minister's Fellowship Scheme for Doctoral Research, co-sponsored by Confederation of Indian Industry, Government of India, and Microsoft Research India.

References

[1] 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*, ser. SOSP '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 19–31. [Online]. Available: https://doi.org/10.1145/74850.74854

[2] B. Verghese, S. Devine, A. Gupta, and M. 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*, ser. ASPLOS VII. New York, NY, USA: Association for Computing Machinery, 1996, p. 279–289. [Online]. Available: https://doi.org/10.1145/237090.237205

[3] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. 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*, ser. ASPLOS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 381–394. [Online]. Available: https://doi.org/10.1145/2451116.2451157

[4] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. 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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 283–300. [Online]. Available: https://doi.org/10.1145/3373376.3378468

[5] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using ksm," in *In OLS*, 2009.

[6] N. Xia, C. Tian, Y. Luo, H. Liu, and X. Wang, "UKSM: swift memory deduplication via hierarchical and adaptive memory region distilling," in *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, N. Agrawal and R. Rangaswami, Eds. USENIX Association, 2018, pp. 325–340. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/xia

[7] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 309–322.

[8] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, p. 181–194, Dec. 2003. [Online]. Available: https://doi.org/10.1145/844128.844146

[9] J.-H. Chiang, H.-L. Li, and T.-c. Chiueh, "Introspection-based memory de-duplication and migration," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 51–62. [Online]. Available: https://doi.org/10.1145/2451512.2451525

[10] A. Panwar, S. 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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–360. [Online]. Available: https://doi.org/10.1145/3297858.3304064

[11] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 705–721.

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

[13] P. Sharma and P. Kulkarni, "Singleton: System-wide page deduplication in virtual environments," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 15–26. [Online]. Available: https://doi.org/10.1145/2287076.2287081

[14] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao, "Cmd: Classification-based memory deduplication through page access characteristics," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 65–76. [Online]. Available: https://doi.org/10.1145/2576195.2576204

[15] L. K. Documentation, "Kernel samepage merging," Online https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html.

[16] B. L. Rodrigo Ceron, Rafael Folco and H. Tsubamoto, "Power systems memory deduplication," Online http://www.redbooks.ibm.com/redpapers/pdfs/redp4827.pdf, 2017.

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

[18] J. Corbet, "Autonuma: the other approach to numa scheduling," Online https://lwn.net/Articles/488709/.

[19] Y. Patel, L. Yang, L. Arulraj, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Avoiding scheduler subversion using scheduler-cooperative locks," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387521

[20] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 89–102. [Online]. Available: https://doi.org/10.1145/1519065.1519076

[21] R. W. Carr, *Virtual Memory Management*. USA: University of Michigan Press, 1984.

[22] A. Panda, "nuksm: Numa-aware memory de-duplication for multi-socket servers," Master's thesis, Indian Insititute of Science, August 2021. [Online]. Available: http://csl.csa.iisc.ac.in/thesis/AkashPanda_2021.pdf

[23] J. R. Tramm, A. R. Siegel, T. Islam, and M. 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. [Online]. Available: https://www.mcs.anl.gov/papers/P5064-0114.pdf

[24] M. Community, "Mysql benchmark tool," Online https://dev.mysql.com/downloads/benchmarks.html.

[25] M. Project, "Btree," Online https://github.com/mitosis-project/mitosis-workload-btree.

[26] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The nas parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, no. 3, p. 63–73, Sep. 1991. [Online]. Available: https://doi.org/10.1177/109434209100500306

[27] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, p. 42–53, May 2008. [Online]. Available: https://doi.org/10.1109/MM.2008.44

[28] 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*, 2013, pp. 59–68.

[29] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao, "Smartksm: A vmm-based memory deduplication scanner for virtual machines," *SOSP Poster*, 2013.

[30] K. Miller, F. Franz, T. Groeninger, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "Ksm++: Using i/o-based hints to make memory-deduplication scanners more efficient," in *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12), London, UK, March 3, 2012*, 2012.

[31] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "XLH: More effective memory deduplication scanners through cross-layer hints," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 279–290. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/miller

[32] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman, "Satori: Enlightened page sharing," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX'09. USA: USENIX Association, 2009, p. 1.

[33] 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*, 2011, pp. 112–120.

[34] A. Garg, D. Mishra, and P. Kulkarni, "Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 44–59. [Online]. Available: https://doi.org/10.1145/3050748.3050760

[35] R. H. 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] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 143–156. [Online]. Available: https://doi.org/10.1145/268998.266672

[37] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–12. [Online]. Available: https://doi.org/10.1145/2830772.2830773

[38] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. 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*. USENIX Association, 2017, pp. 733–744. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/guo-fan

[39] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quema, "Large pages may be harmful on NUMA systems," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 231–242. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/gaud

[40] A. Basu, "Revisiting virtual memory," Ph.D. dissertation, University of Wisconsin-Madison, December 2013.

[41] V. Sri Sai Ram, A. Panwar, and A. Basu, "Trident: Harnessing architectural resources forall page sizes in x86 processors," in *Proceedings of the 54th International Symposium on Microarchitecture*, ser. MICRO-54. New York, NY, USA: ACM, 2021.

[42] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/1972551.1972552

[43] J. Lindemann and M. 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*, ser. SAC '18. New York, NY, USA: Association for Computing Machinery,

2018, p. 183–192. [Online]. Available: https://doi.org/10.1145/3167132.3167151

[44] 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)*, 2013, pp. 1–12.

APPENDIX

A. Abstract

The artifact provides the source of evaluated benchmarks binaries and our Linux modifications. The exact invocation arguments and measurement infrastructure is provided through bash and python scripts which allow reproducing paper results on a two-socket Intel SkyLake (or similar) machine with 380GiB of main memory.

B. Artifact check-list (meta-information)

- **Algorithm:** NUMA-aware Memory De-duplication on Multi-socket Servers
- **Benchmarks:** XSBench, BTree, MySQL, CG, RandomAccess. Source included.
- **Compilation:** GCC version 5.4.0
- **Binary:** Makefile included to compile binaries on x86_64.
- **Data set:** None.
- **Hardware:** We recommend a two-socket Intel Xeon Gold 6140 with 18 cores (36 threads) and 192GiB memory per-socket (384GiB total memory) to reproduce results reported in the paper. Other two-socket x86_64 servers with similar memory and compute capability are expected to produce comparable results.
- **Run-time state:** Populated by the scripts themselves.
- **Execution:** Using bash scripts on a Linux/KVM platform. Scripts, to be executed on the host, require sudo privilege.
- **Output:** The artifact scripts produce a csv file for each figure used in the paper.
- **How much disk space required (approximately)?:** 120GiB
- **How much time is needed to prepare workflow (approximately)?:** 60 minutes
- **How much time is needed to complete experiments (approximately)?:** 80 hours
- **Publicly available?:** Yes
- **Workflow framework used?:** None
- **Archived (provide DOI)?:** 10.5281/zenodo.5139278

C. Description

1) How to access: The artifact is available in the GitHub repository https://github.com/csl-iisc/nuKSM-pact21-artifact as well as at https://doi.org/10.5281/zenodo.5139278.

2) Hardware dependencies: We recommend a two-socket Intel Xeon Gold 6140 with 18 cores (36 threads) and 192GiB memory per-socket (384GiB total memory) to reproduce results reported in the paper. Other two-socket x86_64 servers with similar memory and compute capability are expected to produce comparable results.

3) Software dependencies: The compilation environment, scripts and benchmarks assume Ubuntu 18.04 LTS, which also uses the Linux Kernel v5.4.0. Similar Linux distributions are also expected to work. In addition to the packages shipped with Ubuntu 18.04 LTS, additional packages need to be installed as follows:

```
$ sudo apt-get install build-essential flex \
       libncurses-dev bison libssl-dev \
       libelf-dev libnuma-dev python3 git \
       python3-pip  wget kernel-package gfortran \
       fakeroot ccache libncurses5-dev \
       pandoc libevent-dev libreadline-dev \
       python3-setuptools qemu-kvm virtinst \
       bridge-utils libvirt-bin virt-manager
```

D. Installation

To install, either download the complete artifact from Zenodo or clone the GitHub repository. It contains the source of all benchmarks and scripts required to run the artifact.

1) Compiling binaries: Use the following commands to build binaries on your system:

```
$ cd /path/to/nuKSM-pact21-artifact/
$ git submodule init
$ git submodule update
$ make
```

2) Installing nuKSM kernel: On your test machine, compile and install the vmlinux binary from `./nuKSM-linux/`.

```
$ cd nuKSM-linux/
$ git fetch --all
$ git checkout v5.4
$ cp -v /boot/config-$(uname -r) .config
$ make menuconfig
$ make -j $(nproc)
$ sudo make modules_install
$ sudo make install
$ git checkout nuKSM_SingleTree
$ make -j $(nproc)
$ sudo make modules_install
$ sudo make install
$ git checkout nuKSM_MultiTree
$ make -j $(nproc)
$ sudo make modules_install
$ sudo make install
```

3) Installing and Configuring a Virtual Machine: Install a virtual machine using libvirt on your test machine. An example using command line installation is provided below (choose ssh-server when prompted for package installation). Create a user named nuksm with password nuksm during installation.

```
$ cd VM_images/;

$ virt-install
--name ubuntu_nuksm_1 \
--ram 60000 --vcpus 4 \
--disk path=./ubuntu_nuksm_1.qcow2,size=50 \
--os-type linux \
--os-variant generic \
--network bridge=virbr0 \
--graphics none \
--console pty,target_type=serial \
--location 'http://archive.ubuntu.com/ubuntu/dists/\
bionic/main/installer-amd64/' \
--extra-args 'console=ttyS0,115200n8 serial'

$ virt-clone --original ubuntu_nuksm_1 \
--name ubuntu_nuksm_2 --auto-clone
$ cd -
```

Create a network for the virtual machines:

```
$ cd resources/network_xml/
$ virsh net-define network-01.xml
$ virsh net-start network-01
$ cd -
```

Next, run the following command to generate and load the required VM configurations. This will configure the IP, number and the affinity of vCPUs and memory size of the VMs.

```
$ cd scripts
$ sudo python3 gen_vmconfigs.py 0 1
$ sudo ./load_vmconfigs.sh
```

Once the network is setup and the configurations are loaded, restart the vms.

```
$ virsh shutdown ubuntu_nuksm_1
$ virsh shutdown ubuntu_nuksm_2
$ virsh start ubuntu_nuksm_1
$ virsh start ubuntu_nuksm_2
```

Login to the machines to setup the benchmarks inside the VMs. IPs of ubuntu_nuksm_1 will be 192.168.123.149 and of ubuntu_nuksm_2 will be 192.168.123.228. Perform the following steps on both the VMs to setup the benchamrks and environment.

```
$ ssh nuksm@[IP OF THE VM]
$ sudo apt install net-tools mysql-server
libmysqlclient-dev sysbench git make gcc g++ gfortran
$ sudo systemctl disable mysql
```

Add these lines to /etc/mysql/mysql.conf.d/mysqld.cnf

```
tmp_table_size=20G
max_heap_table_size=20G
```

Switch to the root user to setup mysql user. Run the following commands.

```
# mysql -u root -p

mysql> CREATE USER "nuksm"@"localhost" IDENTIFIED BY
       "nuksm";
mysql> CREATE DATABASE nuksmbench;
mysql> GRANT ALL PRIVILEGES ON nuksmbench.* TO
       'nuksm'@'localhost';
mysql> FLUSH PRIVILEGES;
```

Clone the repository https://github.com/csl-iisc/nuKSM-pact21-artifact in `/home/nuksm/nuKSM-artifact` and compile the benchmarks.

```
$ git clone https://github.com/csl-iisc/
nuKSM-pact21-artifact nuKSM-artifact
$ cd nuKSM-artifact
$ make
```

E. Experiment workflow

Scripts to launch experiments are present in the `./evaluation_script/` directory. Scripts are required to be run by the `root` user.

1) Launching Fairness experiments: Boot the Linux kernel v5.4.0, and run the following scripts.

```
# cd evaluation_script/
# bash run_evaluation_fairness.sh KSM_OFF
# bash run_evaluation_fairness_perf.sh KSM_OFF
# bash run_evaluation_fairness.sh KSM_ON
# bash run_evaluation_fairness_perf.sh KSM_ON
```

Now boot with the Linux Kernel 5.4.0nuKSMSingleTree+ kernel, and run the following scripts.

```
# bash run_evaluation_fairness.sh nuKSM
# bash run_evaluation_fairness_perf.sh nuKSM
```

Note: `run_evaluation_fairness_perf.sh` uses performance counters to get local/remote memory access. It would work only on Intel Skylake machines.

2) Launching priority inversion experiments: Boot from Linux kernel v5.4.0, and run the following scripts.

```
# bash run_priority_inversion.sh KSM_ON
```

Now boot from Linux kernel 5.4.0nuKSMSingleTree+, and run the following scripts.

```
# bash run_priority_nuksm.sh
```

3) Launching scalability experiments: We need to launch these experiments automatically on machine startup to accurately capture CPU utilization of KSM and nuKSM (we use `ps` to record the average CPU utilization). First we re-configure to bind both VMs to node-0 and edit the crontab script:

```
# cd scripts/
# python3 ./gen_vmconfigs.py 0 0
# ./load_vmconfigs.sh
# crontab -e
```

Add the following line to crontab:

```
@reboot /path/to/nuKSM-pact21-artifact/
evaluation_script/crontab_script.sh
```

We need to limit the memory on the machine to 175 GiB in order to run this experiment. To do that, we have to append `mem=175G` to `GRUB_CMDLINE_LINUX` in `/etc/default/grub`.

Reboot the machine and wait for the experiment to finish. It takes around 1600 seconds to get completed. Next, reboot with Linux Kernel 5.4.0nuKSMMultiTree+ and wait for experiments to finish. Now remove the added line from crontab, so that the next reboot will not launch the experiment.

4) Gathering results: To compile results, execute:

```
# cd /path/to/nuKSM-pact21-artifact/scripts/
# bash gather_results.sh
```

This will produce a CSV file for each figure. All results are redirected to `./results/` directory.