AdaptMD: Balancing Space and Performance in NUMA Architectures With Adaptive Memory Deduplication

Lulu Yao, Yongkun Li, *Member, IEEE*, Patrick P. C. Lee, *Senior Member, IEEE*, Xiaoyang Wang, and Yinlong Xu

***Abstract*—Memory deduplication effectively relieves the memory space bottleneck by removing duplicate pages, especially in virtualized systems in which virtual machines run the same OS and similar applications. However, due to the non-uniform access latencies in NUMA architectures, memory deduplication poses a trade-off between memory savings and access performance: global deduplication across NUMA nodes realizes high memory savings, but leads to frequent cross-node remote access after deduplication and results in performance degradations. In contrast, local deduplication avoids remote access, but limits deduplication effectiveness. We design AdaptMD, an adaptive memory deduplication system that addresses the space-performance trade-off in NUMA architectures. AdaptMD leverages hotness awareness to globally deduplicate only cold pages to reduce remote access. It also migrates similar applications to the same NUMA node to allow local deduplication without remote access. We further make AdaptMD readily configurable to address various deployment scenarios. Experiments show that AdaptMD achieves high memory savings as in global deduplication, while achieving similar access performance as in local deduplication.**

***Index Terms*—Memory management, virtual memory, memory deduplication, NUMA architecture, virtualization.**

I. Introduction

NUMA (non-uniform memory access) architectures have been widely adopted by modern server machines in data centers and cloud deployments [2], [12], [23], [32], [41]. By using multiple memory buses and limiting the number of processors in each memory bus, NUMA architectures alleviate memory bus contention and achieve high scalability with dozens of processors in a single server machine [1], [28]. To this end, modern NUMA servers are composed of multiple NUMA nodes (e.g., two or four NUMA nodes [32]), each of which is attached with dedicated memory space. Thus, NUMA servers can provide large memory space and are often used in data centers to support various memory-intensive applications [3], [14], such as graph computing and in-memory data analysis.

However, memory becomes a scarce resource when data centers run a large number of data-intensive applications, as such applications often have large working set sizes [7], [25], [29] and each of them consumes significant memory usage. In particular, when data-intensive applications in virtualized environments, memory consumption becomes even more demanding, as virtual machines (VMs) are often over-provisioned to meet the peak memory demands. As a result, memory is a critical factor that determines the performance of memory-intensive applications. Furthermore, insufficient available memory space can incur frequent memory swaps, which not only significantly degrade the performance of applications [3], [14], [35], but also limit the increase of CPU utilization [16].

Memory deduplication effectively alleviates the memory pressure by eliminating duplicate data and keeping only one physical copy of duplicate pages in memory [4], [17], [40], [42]. It is particularly effective when running applications in virtualized environments, as VMs in the same physical host often run the same OS and similar applications and there exists substantial duplicate data in host memory [15], [32]. Measurement studies show that memory deduplication can achieve 40-50% of memory savings [17], [40]. As memory deduplication is widely supported in modern kernels (e.g., KSM [4] in Linux), it can also be enabled on NUMA servers to eliminate duplicate data.

When memory deduplication is adopted in NUMA architectures, we observe that there exists an inherent trade-off between memory savings and access performance. Conventional memory deduplication adopts *global deduplication* (the default configuration in KSM), which deduplicates the whole memory space of all NUMA nodes. However, if duplicate pages appear in different memory nodes, global deduplication maps duplicate pages to a physical copy in a remote NUMA node. Accessing the memory space of other NUMA nodes has significantly longer latencies than accessing local memory

Manuscript received 15 November 2022; revised 3 February 2024; accepted 21 February 2024. Date of publication 14 March 2024; date of current version 10 May 2024. This work was supported in part by the Youth Innovation Promotion Association CAS. Recommended for acceptance by X. Fu. *(Corresponding author: Yongkun Li.)*

Lulu Yao and Xiaoyang Wang are with the University of Science and Technology of China, Hefei 230026, China (e-mail: luluyao@mail.ustc.edu.cn; wxy1999@mail.ustc.edu.cn).

Yongkun Li and Yinlong Xu are with the University of Science and Technology of China, Hefei 230026, China, and also with Anhui Province Key Laboratory of High Performance Computing, University of Science and Technology of China, Hefei 230026, China (e-mail: ykli@ustc.edu.cn, ylxu@ustc.edu.cn).

Patrick P. C. Lee is with The Chinese University of Hong Kong, 999077, Hong Kong (e-mail: pclee@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TC.2024.3375592

(e.g., 1.5-4×[1], [32]). Thus, global deduplication triggers frequent cross-node memory access, leading to degraded performance. Alternatively, we can adopt *local deduplication* (also supported in KSM), which applies deduplication independently to the local memory region of each NUMA node. This avoids remote memory access, but cannot remove duplicate pages across NUMA nodes. Our experiments on NUMA servers show that global deduplication may suffer from up to 43% performance penalty on a two-node server, and the penalty even reaches up to 60% on a four-node server, while local deduplication only achieves less than 76% of the memory savings of global deduplication (see Section II-C for details).

Although extensive efforts have been made to optimize memory deduplication, they mainly focus on optimizing the execution process of memory deduplication to realize faster and more lightweight deduplication [8], [27], [39], [42]. The problem of incurring frequent remote access and hence longer access frequencies after memory deduplication is not well studied. A recent work, nuKSM [32], optimizes the deduplication performance in NUMA architectures, yet its design is based on global deduplication only. Other studies on NUMA architectures [1], [12], [23] focus on mitigating the impact of asymmetric memory access on application performance, yet they do not consider memory deduplication. Thus, how to realize high memory savings via memory deduplication, while preserving the post-deduplication performance, for NUMA architectures remains an open issue.

To realize efficient memory deduplication in NUMA architectures, particularly on addressing the trade-off between memory savings and access performance, we observe that there are two choices to limit remote accesses. One choice is to leverage hotness awareness, i.e., deduplicating only infrequently accessed "cold" pages across NUMA nodes to avoid frequent remote access; the other choice is to leverage application migration, i.e., migrating "similar" applications that share substantial duplicate pages to the same NUMA node and using only local deduplication to avoid remote access. However, efficiently realizing the above two design choices faces multiple key challenges. First, identifying duplicate pages in all NUMA nodes for cold pages may require many page comparisons and hence incur high performance overhead. Second, intuitive methods of characterizing application similarities (e.g., page-by-page comparison) are time-consuming. Third, migration scheduling must take into account the resource demands of applications and the resource usage of each NUMA node, yet the migration operation should be lightweight to avoid suspending any running applications for a long time. Finally, it is common to deploy various types of applications simultaneously on multiple NUMA nodes [32], [38], [41], and the deployment setting is transparent to the NUMA architectures due to hypervisor abstraction. It is thus difficult to generalize a memory deduplication approach for various application deployment scenarios; instead, the memory deduplication policy should be configurable to adapt to different deployment scenarios.

In this paper, we present AdaptMD, an adaptive memory deduplication system that carefully balances the space-performance trade-off in memory deduplication, such that it achieves high memory savings, while limiting remote memory access. AdaptMD consists of two newly designed memory deduplication techniques, namely AdaptMD-H and AdaptMD-S, as well as an adaptive control module that is readily configurable to choose the deduplication policies to make the best possible space-performance trade-off for different application deployment scenarios. Specifically, AdaptMD-H leverages the idea of hotness awareness for memory deduplication. It applies global deduplication to only cold pages that are infrequently accessed, and applies local deduplication to hot pages to mitigate remote memory access. To make the design efficient, AdaptMD-H uses Bloom filters [6] to quickly locate duplicate cold pages across NUMA nodes, so as to limit the page comparison overhead. On the other hand, AdaptMD-S leverages the idea of migrating similar applications to the same NUMA node for local deduplication. It uses a bitmap-based lightweight scheme to estimate application similarities, and further uses a scheduling scheme based on the application similarities with efficient live migration support.

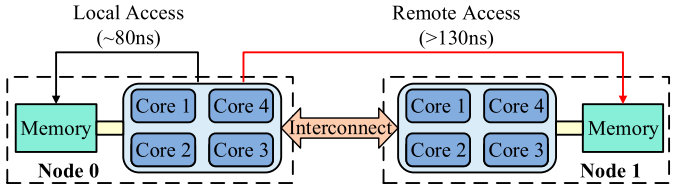


Fig. 1. Overview of a NUMA architecture.

We implement a prototype of AdaptMD atop Linux's KSM [4]. We conduct extensive experiments on NUMA servers through various benchmarks and application deployment scenarios. Experiments show that AdaptMD effectively balances the space-performance trade-off. For example, AdaptMD achieves 93% of memory savings as in global deduplication, while achieving similar access performance as in local deduplication with at most 10% more execution time. We make the artifact of AdaptMD available in the repository https://anonymous.4open.science/r/AdaptMD.

II. BACKGROUND AND MOTIVATION

A. NUMA Architectures

As the number of cores in a single machine keeps increasing, modern data centers deploy NUMA servers to surpass the scalability limits of symmetric multi-processing (SMP) architectures [1], [28]. In particular, NUMA architectures alleviate memory bus contention by limiting the number of processors on one memory bus. In the following, we introduce the memory layout of a NUMA architecture and analyze its performance impact.

Architectural overview. Fig. 1 depicts a simplified architecture of a NUMA server, which consists of multiple NUMA nodes (e.g., two or four NUMA nodes [32]). Each node comprises multiple CPU cores attached with a dedicated memory region. Note that the CPU cores within the same NUMA node share the same memory region, which we refer to as *local memory*. Different NUMA nodes are connected with high-speed

TABLE I
LATENCY (IN NANOSECONDS) OF A CORE IN NODE 0 TO ACCESS ITS LOCAL MEMORY AND REMOTE MEMORY IN NODES 1-3

		Node 0	Node 1	Node 2	Node 3
2-Node	Node 0	82	132	-	-
4-Node	Node 0	76	157	192	200

interconnect components (e.g., Intel UPI and Intel QPI). Thus, each CPU core can also access the memory regions attached to other NUMA nodes. We refer to such memory regions as *remote memory*. We refer to the access to the local memory and remote memory as *local access* and *remote access*, respectively.

Non-uniform access latency. One fundamental feature of NUMA architectures is the non-uniform access latency. Specifically, the remote access latency is much higher (e.g., by 1.5-4× [1], [32]) than the local access latency. The main reason is that remote access needs to traverse cross-node interconnected links and remote memory controllers. To further validate such an effect, we measure the memory access latency on our two-node and four-node NUMA servers using Intel's memory latency checker [19], which evaluates the memory latency when performing pointer chases between different NUMA nodes (see Section IV for our testbed details). Table I shows the results. The remote access latency is 1.61× the local access latency on the two-node server, while the difference further increases to 2.07-2.63× on the four-node server.

B. Memory Deduplication

Memory deduplication is a redundancy elimination technique that saves memory by keeping only one copy of duplicate pages among processes. Many OSes and hypervisors now support memory deduplication (e.g., Kernel Samepage Merging (KSM) in Linux [4] and Transparent Page Sharing (TPS) in VMWare [40]). We focus on KSM in Linux due to its widespread recognition and usage.

Identifying duplicate pages. The core process in KSM is to periodically scan the memory area marked as *deduplicable* and identify duplicate pages by comparing page content. To determine if a scanned page is duplicate, KSM uses a red-black tree to index pages. For each scanned page, KSM compares the scanned page with each page in the red-black tree in a byte-by-byte manner. The scanned page is a duplicate page once the comparison matches; otherwise, it is a unique page. Note that KSM skips frequently updated pages to accelerate deduplication. Specifically, for each unique page, its whole page content is hashed to a checksum. If the checksum differs from the calculated hash in the last scan period, the page content must be changed recently, so it is simply skipped for deduplication.

Page sharing. To realize page sharing after identifying duplicate pages, KSM merges the duplicate pages by modifying the page table. Specifically, KSM keeps only one physical read-only page, and modifies the page table so that the virtual addresses of duplicate pages point to the shared physical page. To write to a shared page, a copy-on-write (COW) operation is triggered, so that the newly copied page can be updated.

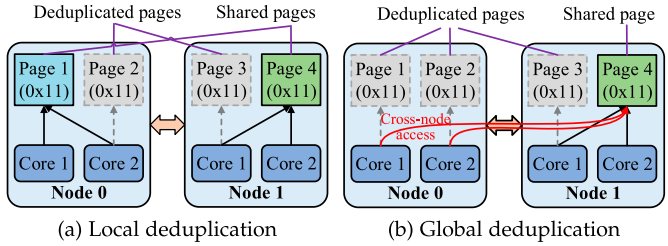


Fig. 2. Memory deduplication in NUMA architectures.

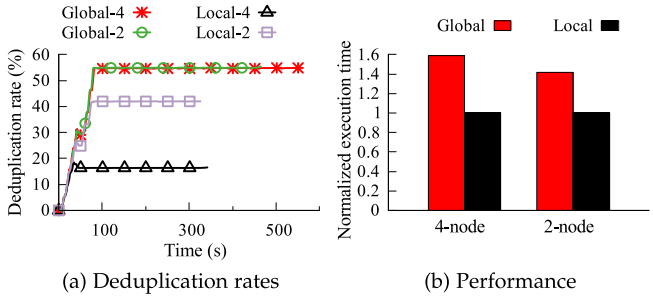


Fig. 3. Trade-off in NUMA architectures. Global/local denotes the global/local deduplication polices, and the number 2 or 4 denotes the number of nodes of the NUMA server.

Deduplication policies in NUMA architectures. Memory deduplication in NUMA servers can run either local deduplication or global deduplication. For example, KSM provides a configurable parameter `merge_across_nodes` to enable global memory deduplication or otherwise (i.e., local deduplication). Fig. 2 depicts the difference of the two deduplication policies. By default, KSM uses global deduplication to maximize memory savings.

C. Trade-Off in NUMA Architectures

Space savings in memory deduplication. We conduct experiments to demonstrate the effectiveness of memory deduplication in space savings. We consider a simple deployment scenario by fixing the number of VMs as four, and deploy the VMs on our four-node and two-node NUMA servers (i.e., the number of VMs running in each NUMA node is one and two, respectively). We assign each VM to one vCPU core and make it run the same application benchmark. We consider four different application benchmarks (see Section IV-B for details). Here, we show only the results of the CG application, which is a benchmark that performs conjugate gradient with irregular memory access and takes around 3 GiB memory, while other application benchmarks show similar conclusions. Fig. 3(a) shows the *deduplication rate*, defined as the ratio of the reduced memory space to the total memory space occupied by all applications. We see that the deduplication rate of global deduplication always exceeds 2× that of local deduplication, while the gain can reach up to 3.4× on the four-node server. We observe a significant difference (24%) in the deduplication rate of Local-4 and Local-2. In Local-4, each node has only one VM, which can only deduplicate the duplicate pages of the VM itself, while in Local-2, each node has two VMs, which can also deduplicate the same pages of different VMs, thereby

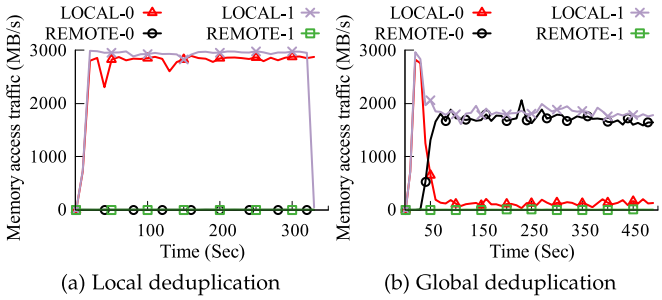


Fig. 4. Memory access traffic per second with local and global deduplication: REMOTE and LOCAL refer to the memory traffic of remote and local accesses, respectively, and the number (0 or 1) refers to the NUMA node from which the accesses are issued.

having a higher deduplication rate. In short, local deduplication has limited memory savings as duplicate data may appear across NUMA nodes, while global deduplication achieves higher memory savings.

Space-performance trade-off. We show via experiments that global and local deduplication policies present an inherent trade-off between memory savings and memory access performance. We use the same experiment setting as in Fig. 3(a), and further show the access performance in Fig. 3(b). We choose the execution time of using local deduplication as the baseline and normalize it to one, and study the increase of the execution time when using global deduplication. We see that global deduplication always incurs longer execution time. It increases the execution time by 42% on the two-node server, and the increase reaches up to 60% on the four-node server. In short, memory access performance with global deduplication can be severely degraded compared to local deduplication.

We have also considered more general settings by deploying more VMs, such as running up to 16 VMs on a four-node server containing 24 CPU cores and deploying a randomly selected application in each VM using the benchmarks we considered; see Section IV-B for details. The results present similar conclusions. Specifically, global deduplication increases the execution time by 11-35%, while local deduplication can only achieve up to 69% of the memory savings compared to global deduplication.

Root cause analysis. The root cause of the performance degradation for global deduplication is the frequent remote access after deduplication. To quantify it, we measure the local and remote memory access traffic using the Intel Resource Director Technology [20]. We still take the CG benchmark as an example. Fig. 4 shows the amount of memory access traffic per second in each NUMA node when running the CG benchmark on the two-node server. We see that local deduplication incurs almost zero remote access. However, for global deduplication, the remote access traffic increases dramatically, for example, up to 2,060 MB/s in node 0. Meanwhile, the local access traffic of node 1 significantly drops (e.g., from 3,000 MB/s to 100 MB/s). The results imply that many duplicate pages originally in node 0 are now mapped to the physical pages in node 1 due to global deduplication, thereby leading to frequent remote memory access.

D. Motivation and Challenges

To address the space-performance trade-off of memory deduplication in NUMA architectures, we have two observations that guide the design of AdaptMD.

First, among the duplicate pages across NUMA nodes, many pages may be infrequently accessed (referred to as *cold* pages). Thus, if we apply global deduplication to cold pages only, it is expected to only degrade performance slightly as remote access is limited. To examine the impact of cold pages, we take the case in Fig. 3 as an example to show the amount of cold duplicate pages. In the four-node case, we see that 6.7 GiB of memory is saved by global deduplication (where the deduplication rate is 54.5% as shown in Fig. 3(a)). Among these memory pages, 25% pages are accessed only once during the whole application runtime. Thus, even if these pages are globally deduplicated, it will not introduce a large amount of remote access. This motivates us to leverage hotness awareness for memory deduplication in NUMA architectures.

Second, it is common to have multiple applications being deployed simultaneously [32], [38], [41], and the application-level deployment is often unaware of the underlying NUMA architecture. Thus, similar applications with substantial duplicate pages may be deployed in different NUMA nodes. This motivates us to leverage VM migration to migrate similar applications to the same NUMA node for local deduplication. In this case, as physical pages are shared by applications running within the same node after deduplication, remote access is limited and hence the performance is maintained.

To efficiently realize the ideas of hotness awareness and VM migration, we address the following design challenges.

- **Lightweight cold page identification and cross-node page lookup.** To differentiate between hot and cold pages, the memory and updating overhead of the metadata used for maintaining the access frequency of pages should be small. Also, as cold pages need to be deduplicated across all NUMA nodes, we need to compare the pages in all nodes to look up duplicate pages. This lookup process should be lightweight with limited computational overhead.
- **Lightweight similarity estimation and VM migration.** The similarities between applications depend on the amount of duplicate pages. Using page content comparison directly to estimate similarities can be expensive as there exist too many pages for comparisons. Also, determining an effective migration needs to accurately estimate the resource usage of each NUMA node, and the migration operation should be lightweight without suspending the applications for a long time.
- **Heterogeneous application deployment.** Different applications may be deployed across NUMA nodes and hence lead to heterogeneous deployment scenarios. The benefits and effectiveness of different deduplication polices often vary depending on the application workloads and the

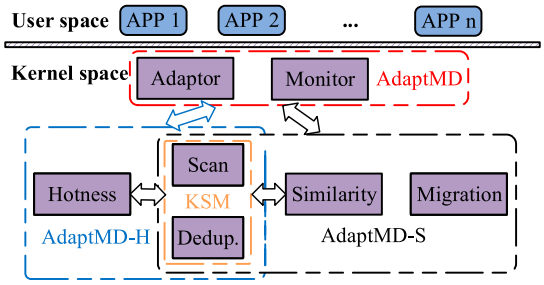


Fig. 5. Overall design of AdaptMD.

deployment of applications in different NUMA nodes. Thus, realizing the best space-performance trade-off necessitates an adaptive scheme to dynamically adjust the deduplication policies.

III. Design of AdaptMD

A. Design Overview

Main idea. AdaptMD is an *adaptive* memory deduplication system that builds on two deduplication techniques, referred to as AdaptMD-H and AdaptMD-S. AdaptMD-H leverages hotness awareness and applies global deduplication to cold pages only to reduce remote access. AdaptMD-S leverages VM migration and migrates similar applications with high content redundancy to the same NUMA node for local deduplication. Finally, AdaptMD adaptively determines the deduplication policies (AdaptMD-H, AdaptMD-S, or the local/global deduplication) depending on the deployment of applications in different NUMA nodes to realize the best space-performance trade-off.

System architecture. Fig. 5 shows the AdaptMD architecture, which adopts a modular design. AdaptMD consists of seven key modules: (i) the *scan module*, which is responsible for scanning the memory pages and recoding the necessary metadata information required by the hotness and similarity modules; (ii) the *hotness module*, which identifies the redundancy for cold pages across NUMA nodes; (iii) the *similarity module*, which measures the similarity between applications; (iv) the *migration module*, which makes the decision of application migration and executes the migration operation; (v) the *monitor module*, which obtains the similarity of applications and the resource occupancy of the system and applications from the *hotness module* and *similarity module*. (vi) the *adaptor module*, which decides the appropriate deduplication choice based on the monitored results; and (vii) the *deduplication module*, which executes the page merging operation and updates the page table for page sharing.

Since the scan and deduplication modules are already provided by KSM, AdaptMD can reuse these modules to limit extra implementation overhead. AdaptMD-H builds on the scan, hotness, and deduplication modules, AdaptMD-S builds on the scan, similarity, migration, and deduplication modules, and the adaptive design builds on the monitor and adaptor modules. In the following, we present the design details of AdaptMD-H (Section III-B) and AdaptMD-S (Section III-C), as well as the adaptive scheme (Section III-D).

B. AdaptMD-H

AdaptMD-H leverages hotness awareness to apply global deduplication to cold pages only. Since cold pages are less frequently accessed, the performance degradation due to remote access to cold pages is limited. AdaptMD-H addresses two key issues: (i) How to differentiate hot and cold pages without incurring significant overhead? (ii) How to quickly identify duplicate cold pages in other NUMA nodes in a lightweight manner?

Hotness identification. To estimate the hotness of each memory page, we define an *inactive counter* for each page, which records the number of successive scan periods in which the page has not been accessed. Specifically, if a scanned page is accessed within two consecutive scan intervals, we reset its inactive counter to zero; otherwise, we increase the inactive counter by one. If the inactive counter reaches a predefined threshold (which we refer to as the *coldness threshold*), we tag the scanned page as cold.

AdaptMD-H determines whether a page is accessed by checking its access bit. The access bit indicates whether a scanned page has been accessed since it is checked in the last scan period. The bit is one if it has been accessed; or zero otherwise. In particular, the scan module periodically scans all memory pages in the memory area marked as *duplicatable* and checks the access bit of each scanned page by calling the `page_referenced` function in Linux kernel.

We argue that AdaptMD-H is lightweight. Since the original deduplication workflow in KSM has already used a background thread to periodically scan memory pages, we can reuse the background thread for our scanning to limit extra overhead. By doing so, the only additional work required by AdaptMD-H is to update the inactive counter when each page is scanned and tag the page as cold if needed, so the computational overhead is small. In terms of extra memory overhead, the inactive counter needs one byte per page in our implementation, and it is stored together with other existing deduplication metadata in KSM (including the virtual address, checksum, positions in the comparison trees, etc.). Thus, the extra memory overhead is small compared with the original metadata in KSM, which has more than 100 bytes in total for each page. In short, AdaptMD-H incurs limited extra overhead.

Cross-node lookup of duplicate cold pages. Recall that AdaptMD-H needs a lightweight way to identify duplicate pages in other NUMA nodes for each cold page. AdaptMD-H deploys Bloom filters [6] to reduce unnecessary page comparisons and accelerate the page lookup process. Fig. 6 depicts the Bloom filter structure. Specifically, AdaptMD-H creates a Bloom filter for each NUMA node. Each bloom filter is a one-dimensional vector whose size is proportional to the number of pages per NUMA node, and each bit corresponds to a 4 KiB page.

AdaptMD-H constructs a Bloom filter coupled with two-level hashing in each NUMA node in a lightweight

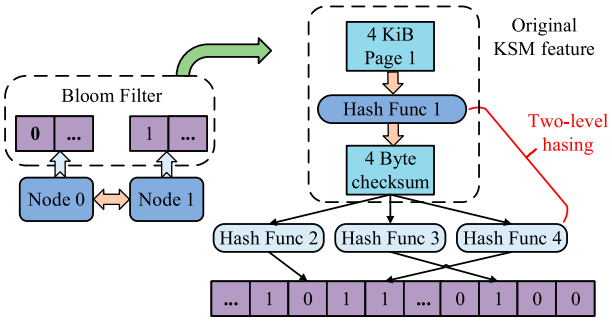


Fig. 6. Bloom filter construction with two-level hashing in each NUMA node.

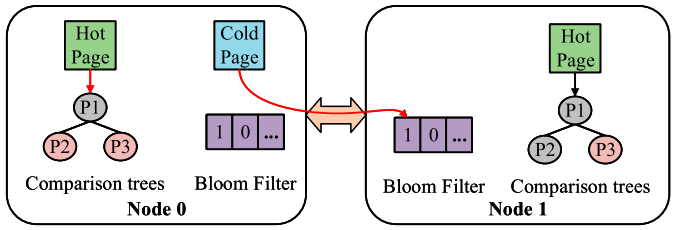


Fig. 7. Deduplication process of AdaptMD-H.

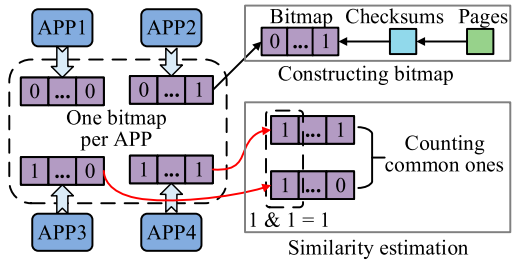


Fig. 8. Similarity estimation in AdaptMD-S.

manner. Specifically, for each page in a NUMA node, the first-level hashing uses only *one* single hash function to compute a 32-bit checksum from the page content. The second-level hashing uses *multiple* independent hash functions, but computes the hash values from the first-level hashed checksum instead of the original page to reduce the computation overhead. We further use the 32-bit checksum as the input to the second-level hash functions to compute the bit positions of the Bloom filter to be set to one.

Considering that the range of values represented by the 32-bit checksum is much larger than the number of bits in a Bloom filter, we can use simple bit-wise operations to implement the second-level hash functions. In particular, AdaptMD-H performs bit-wise AND operations on the checksum and Bloom filter length, and performs bit-wise AND operations on the Bloom filter length after shifting right by 3 and 6 bits to generate three different secondary hash values. Finally, we set the corresponding bits in the Bloom filter as one based on the second-level hashes. To limit the false positive rate of the Bloom filters, AdaptMD-H uses three hash functions in the second-level hashing and sets the size of each Bloom filter (in bits) as eight times the total number of pages in each NUMA node. In this case, the false positive rate of the Bloom filter is 3.06% [11], while the size of the Bloom filter is 32 MiB for 128 GiB memory with 4 KiB pages.

The cross-node lookup procedure based on Bloom filters is as follows. For a cold page, AdaptMD-H checks if a duplicate page has been stored in other NUMA nodes by issuing a lookup to the Bloom filter in each of the other NUMA nodes. If the lookup to a NUMA node returns false, the page must not exist in that node and AdaptMD-H checks another NUMA node; otherwise, if the lookup returns true, AdaptMD-H further checks the comparison trees (i.e., red-black trees) to confirm if the duplicate page really exists, as the Bloom filter may have false positives.

The Bloom filter construction with two-level hashing incurs only low computational overhead. In particular, as a checksum is also needed by the original deduplication module in KSM (Section II-B), we reuse the checksum as the first-level hash result to reduce the extra hashing overhead. For the second-level hashing, we only compute three hashes from a 4-byte checksum, so it is very efficient. For example, it reduces 16% computational overhead compared with computing three hashes from the 4 KiB page as studied in our experiments (Section IV-B).

Hotness-aware deduplication. Fig. 7 depicts the deduplication process of AdaptMD-H. Specifically, for each cold page, if an identical page is found by checking the Bloom filters, we execute the page compare-and-merge process in the deduplication module to realize global deduplication. Otherwise, it is skipped for global deduplication just as hot pages, and we run local deduplication using the original deduplication in KSM.

C. AdaptMD-S

AdaptMD-S migrates similar applications that contain a large fraction of duplicate pages to the same NUMA node, and then performs local deduplication so as to avoid frequent remote access after deduplication. It addresses three key challenges: (i) How to accurately estimate the similarity of any pair of applications? (ii) How to decide the appropriate migration scheduling scheme to reduce the migration overhead? (iii) How to support efficient migration of the application process, including all memory pages and the page table?

Similarity estimation. AdaptMD-S uses bitmaps to estimate the similarity between applications. Fig. 8 shows the main idea. Specifically, AdaptMD-S creates a bitmap for each application. Based on the bitmaps of two applications, AdaptMD-S defines the similarity as the ratio of the number of common ones in both bitmaps (i.e., the bits at the same position are both one) to the minimum number of memory pages in both applications. A larger similarity implies more duplicate pages between applications.

To make the generation of the bitmaps lightweight, we reuse the checksum again needed by the original deduplication in KSM (Section II-B). Specifically, for each scanned page in an

application, as the checksum is already computed for checking the change of page content, we simply update the bitmap by setting the corresponding bit as one according to the checksum. In particular, we set the corresponding bit by using the bit-wise AND operation between the 32-bit checksum and the length of the bitmap, which is also used in the second-level hashing of AdaptMD-H. After scanning all pages in an application, the bitmap is also generated.

Note that the bitmap-based similarity estimation is lightweight and efficient. First, as the construction of the bitmap reuses the checksum generated by KSM, the CPU overhead is negligible. The memory overhead of the bitmap is also small, as each 4 KiB page is just represented by one bit (e.g., a 1 MiB bitmap can represent 32 GiB memory pages). Second, it is efficient to estimate the number of duplicate pages between any two applications. For example, if two applications have a high similarity, then they have a lot of duplicate pages with a high probability as the duplicate pages will set the same bitmap position in both bitmaps as one. However, we emphasize that our similarity estimation is only an approximation, as hash collisions may wrongly increase the similarity. Also, the bitmaps fail to record the frequency of multiple duplicate pages.

Migration scheduling. After determining the similarity between any pair of applications, AdaptMD-S needs to address two issues: (i) which application to migrate, (ii) which NUMA nodes as the target nodes for each migration.

For the first issue, AdaptMD-S chooses applications whose similarity exceeds a predefined threshold (which we refer to as the *similarity threshold*) to migrate. Specifically, AdaptMD-S groups applications whose pair-wise similarities exceed the similarity threshold to form a candidate group. Then it migrates applications in the same candidate group to the same NUMA node. Note that the similarity threshold indicates the lower bound on the number of duplicate pages between the applications in the candidate group. Furthermore, we set the default similarity threshold as 40% based on the experimental analysis of the applications mentioned in Section IV-A.

For the second issue, AdaptMD-S selects the appropriate NUMA nodes based on their priority and resource usage. First, we set the priority according to the number of applications that are in the candidate group and already running in the NUMA node. A higher number of applications implies a higher priority for the NUMA node. Note that the priority of a NUMA node may vary across candidate groups. Second, AdaptMD-S judges whether the NUMA nodes have sufficient memory and CPU resources one by one in a descending order of their priorities. To check whether a NUMA node has enough resources, AdaptMD-S compares the available memory and cores in the NUMA node with the required memory and cores of the applications to be migrated. If a NUMA node has sufficient memory and CPU to run all applications in the candidate group, it is selected as the target node for migration.

We emphasize that the migration scheduling of AdaptMD-S is performance friendly. Note that migrating an application from one NUMA node to other nodes causes cache misses, TLB miss, etc., which in turn degrade application performance. On the one hand, AdaptMD-S filters out applications with low similarity through a predefined similarity threshold to reduce unnecessary migrations. On the other hand, determining candidate target nodes based on priority minimizes the number of migrated applications.

Application migration across node. After determining the applications to be migrated and the target NUMA node, AdaptMD-S needs to perform a specific migration operation and perform local deduplication within the node after migration. For the migration operation, AdaptMD-S needs to migrate three types of data: (i) the application process, (ii) all memory pages, (iii) the page table.

AdaptMD-S supports cross-node live migration of applications without suspending them, so as to minimize the migration overhead. Specifically, for an application process, AdaptMD-S simple calls the function `sched_setaffinity()` in the Linux kernel to set the CPU affinity of the application. One subtle issue is that we need to set the affinity for each thread if an application consists of multiple threads. Otherwise, there are still some threads issuing remote accesses. For example, in a virtualized environment, migrating the VM process does not really migrate the applications running in the VM, so we have to scan the application threads and set affinity for each thread.

For memory pages, AdaptMD-S differentiates shared pages and unshared pages. For unshared pages, it directly migrates the page to the target node by calling `migrate_pages()`. For shared pages, AdaptMD-S breaks them with a copy-on-write (COW) operation, and allocates the copied pages directly in the target node without migration. The reason why AdaptMD-S uses COW is that for applications not being migrated, they may have remote memory accesses to the migrated shared pages, if the migration of shared pages is simply executed by calling page migration function.

For the page table, AdaptMD-S also migrates it to avoid cross-node page table access. Note that the kernel does not support page table migration, and cross-node page table access also hurts application performance [1], [33]. To migrate the page table of an application, AdaptMD-S traverses the multi-level page table and then performs the migration.

We emphasize that AdaptMD-S does not execute continuously like AdaptMD-H and other memory deduplication schemes will be executed after AdaptMD-S executes successfully. Specifically, after executing AdaptMD-S, highly similar applications will be migrated to the same NUMA node, and global deduplication will be performed to eliminate same pages in the system. In our current implementation, AdaptMD will trigger AdaptMD-S again after a user-specified scan period.

D. Adaptive Design

Different deduplication policies pose different trade-offs between memory savings and access performance in different application deployment scenarios. Specifically, global deduplication can also have good performance (or has limited remote accesses) when applications in different nodes have a

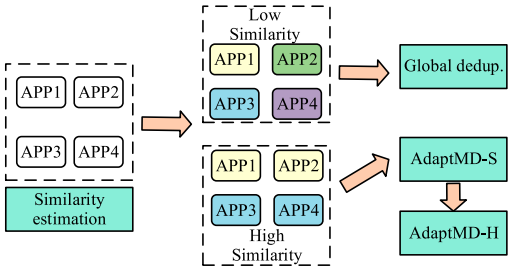


Fig. 9. Adaptive design of AdaptMD.

small similarity, e.g., in the scenario where the applications running in different NUMA nodes are completely different. However, if there are highly similar applications in different nodes, then AdaptMD-S realizes a better trade-off if the similar applications can be migrated to the same node. In other cases, AdaptMD-H should be a better choice, as it achieves higher memory savings than local deduplication and introduces no performance degradation compared with global deduplication. To this end, we propose an adaptive scheme to select the most appropriate deduplication policy according to the deployment scenario, so as to realize the best possible space-performance trade-off in different scenarios.

Adaptive scheme. The workflow of the adaptive scheme is as follows. The monitor module periodically fetches the similarity results from the similarity module and monitors the resource usage of each NUMA nodes. The adaptor module then determines the deduplication polices based on the information from the monitor module. It informs the hotness or migration modules the decision result so as to execute the specific deduplication policy.

Fig. 9 shows the process of selecting a specific deduplication policy. Specifically, if there exist no similar applications in different NUMA nodes, i.e., the estimated similarity is smaller than the predefined similarity threshold, then global deduplication is adopted. Otherwise, we check the resource usage of all NUMA nodes and examine whether the resource requirements of similar applications can be satisfied. If there exists a NUMA node to satisfy the requirements, we execute AdaptMD-S to achieve high memory savings; otherwise, we deploy AdaptMD-H to avoid performance degradation. Note that there should not exist similar applications across nodes after migration by using AdaptMD-S, so we execute global deduplication after the migration. As AdaptMD-H outperforms local deduplication in both aspects of memory savings and access performance with negligible overhead, so we do not use local deduplication as an option in the adaptive design.

Remarks. AdaptMD can automatically determine the best deduplication policy based on the system state and application deployment scenarios. The adaptive decision is lightweight and incurs limited extra overhead, as the required information for the decision making in the adaptive design is obtained from the outputs of other modules. AdaptMD can also be *manually* configured if the information about the application deployment is known in advance. For example, if we know that same applications are deployed in different NUMA nodes, then AdaptMD-S should be enabled to migrate similar applications to the same node; if the migration fails, (e.g., there is no enough resource to run the applications in the same node), then AdaptMD-H should be used to avoid frequent remote access.

E. Discussion

Heterogeneous NUMA nodes. The increase in memory types (e.g., Non-volatile memory, high bandwidth memory) has made NUMA architectures more heterogeneous. Nonetheless, AdaptMD can still be of value, mainly due to the design idea of AdaptMD to minimize memory accesses across NUMA nodes and save more memory. For heterogeneous NUMA nodes based on hybrid memory, AdaptMD-H distinguishes between hot and cold data in memory for deduplication, and AdaptMD-S saves more memory by migrating highly similar applications to the same node. Note that there are some drawbacks in applying AdaptMD to heterogeneous NUMA nodes. First, AdaptMD ignores the characteristics of these memory media, such as persistence, bandwidth, etc., which can be considered by integrating AdaptMD and data placement policies. Second, AdaptMD periodically scans memory for deduplication based on application similarity. However, in heterogeneous NUMA nodes, whether data movement is between NUMA nodes or memory media needs to be considered more carefully.

Page size and type. By default, AdaptMD splits huge pages for deduplication and handles only anonymous pages as it is implemented atop KSM. It first aggressively splits anonymous large pages (e.g., 2 MiB pages) into base pages (e.g., 4 KiB pages), and then performs deduplication in the units of base pages. Note that AdaptMD can also seamlessly work with different page types and page sizes to support large-page-friendly deduplication (e.g., Ingens [22] and SmartMD [15]) as well as deduplicating other types of pages (e.g., file pages), by replacing the underlying deduplication module in KSM accordingly.

OS-level process migration. The deduplication effectiveness in AdaptMD may be affected by user-triggered process migration. In particular, modern OSes allow users to issue process migration; for example, Linux uses `sched_setaffinity()` to specify the CPU core and `migrate_pages()` to migrate memory. However, the migration policy specified by users may be different from the migration decision in AdaptMD, thereby affecting the deduplication effectiveness of AdaptMD.

Complex concurrent applications. Concurrent queries [31] may have negative impact on performance, as they can cause more severe memory access contention for memory pages being deduplicated and shared, thus increasing the chance of CoW and degrading memory access performance. We point out that this performance impact is usually unavoidable as long as deduplication is enabled. Furthermore, the impact of AdaptMD should be the same as that of the conventional deduplication policy such as KSM as long as they have the same deduplication rate. In fact, there is a tradeoff between memory saving and performance when deploying deduplication. We emphasize that

TABLE II
APPLICATION BENCHMARKS

Benchmarks	Description	Parameter Configurations	Memory Usage
Graph500 [13] (abbrv. Graph)	It is a data-intensive HPC benchmark that performs breadth-first search over a graph to complement the Top 500 supercomputers.	breadth-first search with a scale of 21 and edgefactor of 10.	2.90 GiB
StreamCluster [37] (abbrv. SC)	It is a data mining benchmark from PARSEC/Rodinia that solves the online clustering problem.	The number of data points processed is 565536, other parameters use the default value.	2.76 GiB
CG [30]	It is a benchmark that performs conjugate gradient, irregular memory access, and communication to evaluate the performance of supercomputers.	The C class of openmpi and the problem size is 150000.	3.06 GiB
Liblinear [24] (abbrv. Lib)	It is a benchmark of large-scale linear classification for data with millions of instances and features; we train the file *YearPredictionMSD*	L2-regularized L2-loss support vector regression (dual) (i.e., -s 12)	3.46 GiB

AdaptMD focuses on the space-performance tradeoff that exists for memory deduplication in NUMA architectures, aiming at choosing the appropriate deduplication policies in different application scenarios.

IV. EVALUATION

We implement a prototype of AdaptMD atop KSM in Linux kernel v4.4. The prototype itself contains around 2,500 LOC. As AdaptMD builds on the original deduplication module in KSM, it deduplicates only the anonymous pages like KSM.

A. Setup

Testbed. Our experiments run on a NUMA server with 125 GiB of memory. The server has two physical NUMA nodes that are each split into two NUMA nodes via Cluster-On-Die [18], so it is configured with four NUMA nodes for evaluation. Each NUMA node has a Xeon E5-2650 v4 2.2GHZ CPU with six CPU cores. The local and remote memory access latencies are listed in Table I (Section II-A). To better demonstrate the effectiveness of memory deduplication, we deploy applications in VMs by running experiments with QEMU and KVM. By default, we boot up four VMs in each NUMA node (i.e., a total of 16 VMs in the server). Each VM is assigned one vCPU and 4 GiB of RAM, and both the host and guest OSes are Ubuntu 16.04. We bind each vCPU to a physical CPU core.

For the OS of each VM, we keep its factory configuration. Also, we disable the *transparent hugepages* feature in Linux kernel of the host OS, so as to eliminate the impact of huge pages (Section III-E). Note that the benchmarks run in the VM and the VMs run on the host. AdaptMD is implemented based on the KSM module of the host OS kernel. In other words, we do not modify the VM OS and the hypervisor of the host, only the OS of the host.

Application benchmarks and deployment. We consider four application benchmarks that are also used in the evaluation by prior memory deduplication work [15], [32], [34], [44]. Table II presents the benchmarks. We run one application benchmark in each VM and start all VMs to run their benchmarks at the same time. Note that the same benchmark always uses the same input and dataset. We also simulate different scenarios by considering different VM deployments, so as to show the robustness of AdaptMD. In particular, we consider three different deployment scenarios:

- **Mirror.** Each NUMA node runs four different applications in the four VMs. We replicate and run the same set of applications across all NUMA nodes.
- **Random1.** For each of the 16 VMs, we randomly select an application to run on each VM. The sets of application benchmarks running in Nodes 0-3 in the resulting layout are {Graph, Graph, Lib, CG}, {Lib, Lib, CG, CG}, {Graph, Graph, SC, SC}, {SC, SC, Lib, CG}, respectively.
- **Random2.** It is also a random deployment as Random1. The sets of application benchmarks running in Nodes 0-3 are {SC, SC, Lib, Lib}, {SC, SC, Lib, CG}, {Graph, Lib, CG, CG}, {Graph, Graph, SC, Lib}, respectively.

Comparison baselines. Local deduplication and global deduplication represent the two extreme points in the design space of memory deduplication in NUMA architectures, and we treat them as baselines in our evaluation. Note that both policies and AdaptMD run atop KSM for fair comparison. We also take global deduplication with UKSM [42], which is the state-of-the-art deduplication scheme, as a baseline to show that existing optimizations for memory deduplication still face the space-performance trade-off in NUMA architectures. In addition, we compare AdaptMD with nuKSM [32], which aims to optimize the existing NUMA-unaware memory deduplication strategies. In particular, nuKSM includes two optimizations: one for NUMA-awareness to avoid priority subversion and guarantee fairness by deciding the placement of deduplicate pages, named *nuKSM-SingleTree*, and the other for scalability in large memory spaces by replacing the original two centralized comparison trees with two forests, named *nuKSM-MultiTree*. We emphasize that both nuKSM-SingleTree and nuKSM-MultiTree are optimization strategies based on global deduplication.

Parameters. By default, we set `pages_to_scan` (i.e., the number of pages being scanned in each scan period) as 100,000. Since we focus on the performance after deduplication, we set `sleep_millisecs` as zero to run deduplication as fast as

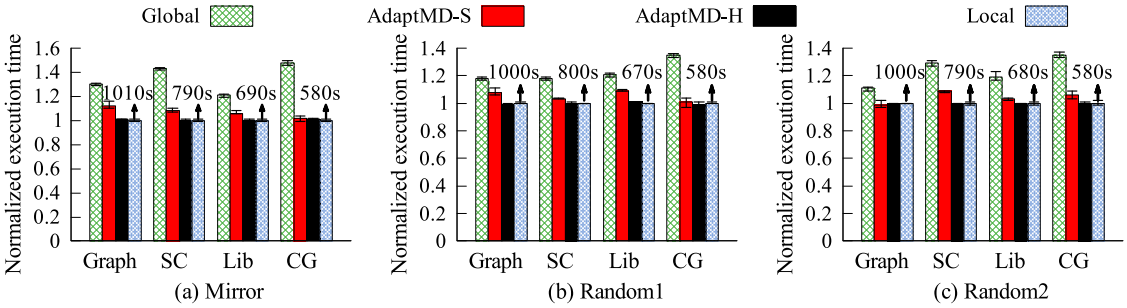


Fig. 10. Exp#1: performance in different scenarios.

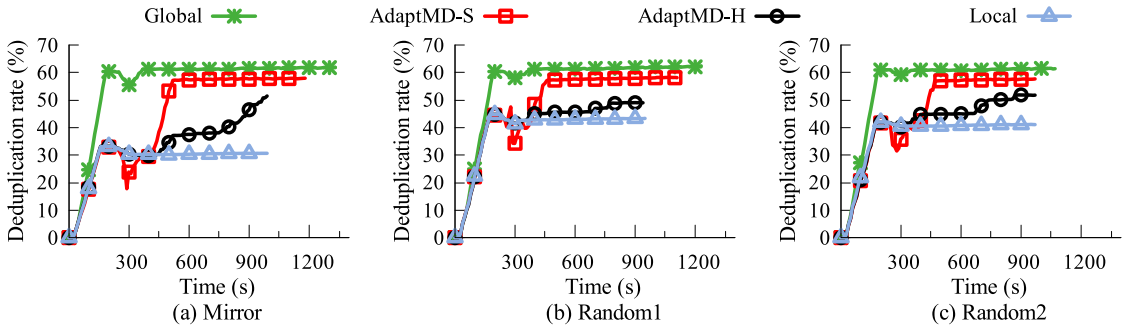


Fig. 11. Exp#1: deduplication rate in different scenarios.

possible. Note that the two parameters affect the deduplication rate as well (e.g., the same content pages may not be found in a short time duration if the deduplication is too slow). In addition, AdaptMD has two configurable thresholds: (i) the coldness threshold (Section III-B), which we set as one by default; and (ii) the similarity threshold (Section III-C), which we set as 40% by default. We also study the sensitivity by varying the two configurable parameters.

Performance metrics. We use the execution time of each application benchmark after deduplication is completed to quantify the VM performance and use deduplication rate to evaluate memory savings. For each experiment, we run six times and show the average results. In addition, we normalize the execution time of local deduplication as one for ease of presentation. We compute the deduplication rate as the ratio of the number of saved pages to the number of scanned pages. In particular, we obtain the number of pages saved by deduplication (i.e., the value of `pages_sharing` in KSM) and the total number of pages being scanned (i.e., the sum of `pages_sharing`, `pages_shared`, `pages_unshared`, and `pages_volatile` in KSM) from the deduplication module every one second in the host.

B. Performance Results

Experiment 1: Space-performance trade-off. We compare AdaptMD-H and AdaptMD-S with existing global and local deduplication policies to study the trade-off. Figs. 10 and 11 show the performance and memory savings, respectively. We run each experiment six times, and also show the error bars for the performance results in Fig. 10. As the variance between the results of multiple runs is very small, we show only the average results in later experiments.

First, the trade-off indeed exists in NUMA architectures for different VM deployments. Specifically, the execution time under global deduplication with mirror deployment increases by up to 48%, and can still increase by 11-35% even in random deployments. Also, global deduplication always achieves a much higher deduplication rate than local in all deployments; for example, the deduplication rate increases from 30% to 60% in the mirror deployment. In particular, the deduplication rate of local deduplication reaches only 50%, 69% and 67% of global deduplication under the mirror, random1 and random2 deployments, respectively.

Second, AdaptMD-H and AdaptMD-S achieve a better balance on the space-performance trade-off. Specifically, AdaptMD-H achieves almost the same performance as local deduplication in all VM deployment scenarios. This implies that the hotness-aware scheme in AdaptMD-H effectively avoids frequent remote access.

Furthermore, we measure the number of CoW pages of the deduplicated cold pages. Specifically, we calculate the percentage of copy-on-write (CoW) pages in cold pages that are globally deduplicated, and the results show that the percentage is always less than 0.1% in different deployment scenarios. In particular, about 1500 pages undergo CoW in the mirror deployment, and the number of deduplicated globally cold pages reaches 1 800 000, significantly reducing the impact of AdaptMD-H on application performance. The above experimental results demonstrate the validity of the hotness

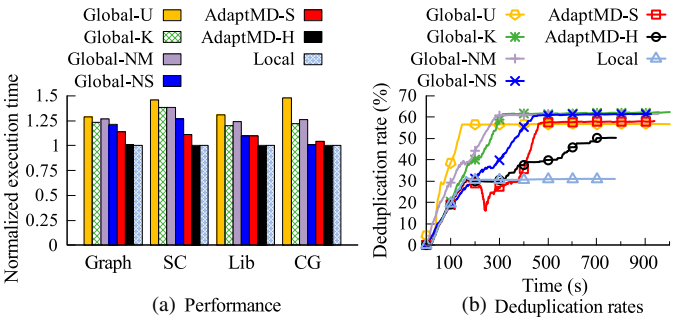


Fig. 12. Exp#2: comparison with UKSM and nuKSM.

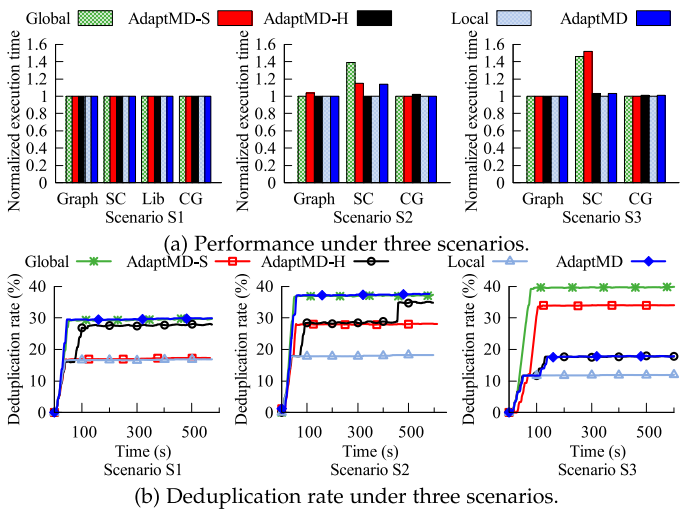


Fig. 13. Exp#3: effectiveness of the adaptive scheme.

identification of AdaptMD-H. Meanwhile, it achieves up to 20% higher deduplication rate than local deduplication by deduplicating cold pages globally. On the other hand, AdaptMD-S realizes a very similar deduplication rate with global deduplication, with a difference of no more than 4% in all cases. Meanwhile, AdaptMD-S also significantly reduces the performance loss compared with global deduplication; e.g., the execution time increases by 0%-14%. AdaptMD-S has performance loss since the memory access may be influenced by lock contention and cache/TLB misses during VM migration.

Experiment 2: Comparison with UKSM and nuKSM. We replace the default deduplication module KSM with two optimized designs UKSM and nuKSM for the global deduplication policy. We denote the UKSM-based global deduplication as Global-U, and denote the nuKSM-SingleTree-based and nuKSM-MultiTree-based global deduplication as Global-NS and Global-NM, respectively, and refer to the original global deduplication in KSM as Global-K. We also include the baseline of local deduplication in the experiment. As UKSM aims to improve deduplication performance (e.g., faster deduplication and lower CPU overhead), right after deduplication is enabled, we immediately start the application. Thus, the application runs concurrently while deduplication proceeds. We set the maximum percentage of occupied CPU cycles (`max_cpu_percentage`) used by UKSM as 100% and the sleep time (`sleep_millisecs`) between consecutive scans as zero, so as to maximize the deduplication performance. nuKSM also uses the same experimental setup as KSM and runs applications and deduplication process concurrently.

Fig. 12 shows the results under mirror deployment. We omit the results for other deployments as we observe similar results. We can see that global deduplication polices, including Global-U, Global-NS and Global-NM, all have the space-performance trade-offs. Specifically, Global-U achieves almost the highest memory savings (e.g., a deduplication rate of 57%), but always incurs the largest performance loss (e.g., 29-48% increase in execution time). The reason is that it applies global deduplication, so it still suffers from the space-performance trade-off, similar to Global-K. Also, speeding up the deduplication process causes duplicate pages to be removed earlier, so Global-U suffers from a higher performance degradation due to the increased amount of remote accesses. For example, under the CG workload, Global-U incurs 26% more execution time than Global-K. Since UKSM achieves faster deduplication speed, so its deduplication rate also increases at a faster rate, as shown in Fig. 12(b). Global-NM is similar to Global-U in that it has better deduplication responsiveness due to the use of multiple comparison trees, and therefore has faster deduplication speeds than Global-K, which ultimately has a greater performance loss. Compared to Global-K, Global-NS deduplicates duplicated pages more slowly, for example, Global-NS takes extra 130 seconds to reach a stable deduplication rate compared with Global-K, which makes it have smaller performance degradation (e.g., 1-27% execution time increment). In short, this experiment shows that existing optimizations on KSM still cannot effectively address the space-performance trade-off in NUMA architectures.

Experiment 3: Effectiveness of the adaptive scheme. To show the effectiveness of the adaptive scheme, we deploy one VM per NUMA node, and consider three VM deployment scenarios: (i) *Scenario S1: Low redundancy*, in which the four VMs run four different applications (i.e., Graph, SC, Lib and CG); (ii) *Scenario S2: High redundancy and low resource requirement*, in which two VMs run the same application SC and the two VMs run Graph and CG; and (iii) *Scenario S3: High redundancy and high resource requirement*, in which the deployment is the same as in Scenario S2, while the number of threads and data size used by SC and the number of vCPUs in a VM are scaled up by 5× to simulate a high resource requirement setting.

The experimental results are shown in Fig. 13. Note that local deduplication is not taken as an option in the adaptive design (Section III-D), and AdaptMD refers to our adaptive scheme. We can conclude that AdaptMD always chooses the most appropriate deduplication policy in all scenarios. Specifically, in scenario S1, global deduplication realizes the best trade-off, as there is no much redundancy across all NUMA nodes and global deduplication does not incur frequent remote accesses.

In scenario S2, AdaptMD first chooses AdaptMD-S to migrate highly similar applications to the same NUMA node, and

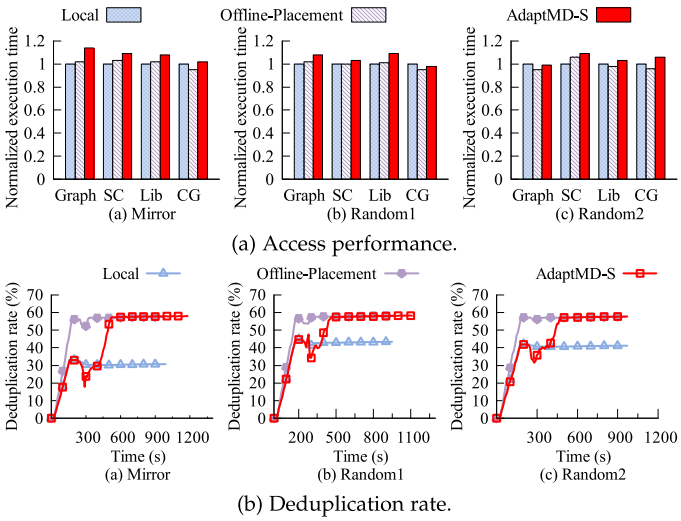


Fig. 14. Exp#4: effectiveness of the similarity estimation.

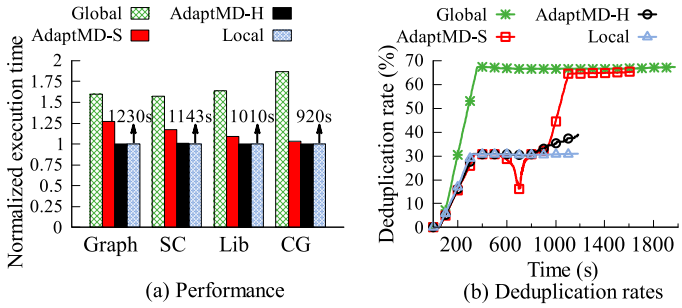


Fig. 15. Exp#5: trade-off on an eight-node NUMA machine.

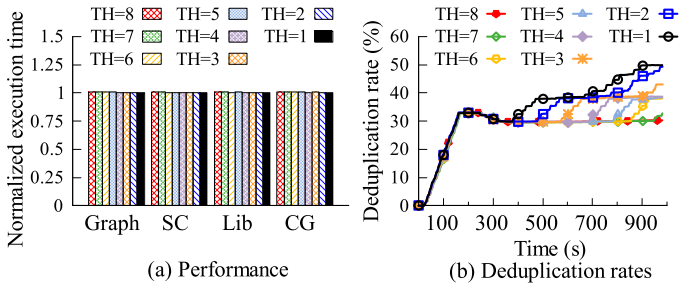


Fig. 16. Exp#6: impact of coldness threshold (abbrv. TH).

then uses global deduplication to eliminate redundant pages across NUMA nodes, thus achieving a higher deduplication rate than AdaptMD-H and AdaptMD-S. Specifically, since there are highly similar applications and sufficient resources on each NUMA node, AdaptMD chooses AdaptMD-S as the initial deduplication strategy. After executing AdaptMD-S, there are no highly similar applications across NUMA nodes, and global deduplication can save more memory at this time. In addition, the deduplication rate of AdaptMD-H in S2 has a sudden increase of around 480s, which is due to the termination of the SC, and all pages become cold and are globally deduplicated by AdaptMD-H. AdaptMD-H performs better in S3 as the resources are not sufficient to support application migration.

Experiment 4: Effectiveness of the similarity estimation. We now examine the effectiveness of the similarity estimation in AdaptMD-S by comparing it with an idealized offline placement scheme. Specifically, the offline scheme is assumed to have an oracle to know the accurate similarity values, and we manually place the similar VMs within the same NUMA node offline before running the applications. In particular, the accurate similarity value between two VMs is calculated from the deduplication rate obtained after deduplicating the two VMs. For example, two VMs with a deduplication rate of 40% means that 80% of the content of a VM is the same as that of other VM, so the similarity value is 80%.

Fig. 14 show the access performance and deduplication rate. We also consider different VM deployment scenarios as before. We observe that AdaptMD-S has very close access performance compared with the idealized placement scheme (i.e., Offline-Placement in Fig. 14) under different VM deployment scenarios. Furthermore, for memory savings, due to similarity-based migrations, AdaptMD-S eventually realizes the same deduplication rate. Note that the offline placement scenario realizes the optimal space-performance trade-off, so our comparison results justify the effectiveness of the similarity-based migration scheme. Fig. 14 also shows that the performance difference between offline placement and local deduplication is within 5%. For some applications (e.g., CG), offline placement outperforms local deduplication, mainly because deduplication saves lots of memory, ultimately reducing TLB and cache contention.

Experiment 5: Impact of the number of NUMA nodes. We also evaluate AdaptMD on an eight-node NUMA machine, which contains 256 CPU cores. In the interest of space, we present only the results under the mirror deployment (Section IV-A), and Fig. 15 shows the results. We have similar conclusions as in the four-node setting shown in Experiment 1. Global/local deduplication presents a space-performance trade-off. AdaptMD-S balances the trade-off. It eventually achieves 96% of memory savings of global deduplication, and also reduces the execution time by 33-84% for different applications compared to global deduplication. In this specific scenario, AdaptMD-H has a similar performance with local deduplication. It achieves almost the same performance, and slightly increases the deduplication rate by 9%. In addition, we also observe that the space-performance trade-off for global/local deduplication is more severe when using the eight-node server. This is because when there are more NUMA nodes, more VMs issue remote memory accesses after global deduplication, and thus leading to larger performance degradation.

Experiment 6: Impact of the coldness threshold. Fig. 16 shows the results when the coldness threshold is varied from one to eight. AdaptMD-H has almost the same performance, while it has a larger deduplication rate for smaller coldness threshold. The reason is that due to the long scanning period, a page that is not accessed in one period is indeed cold, so setting the threshold as one already serves as a conservative choice in terms of the performance. Setting a larger threshold treats less

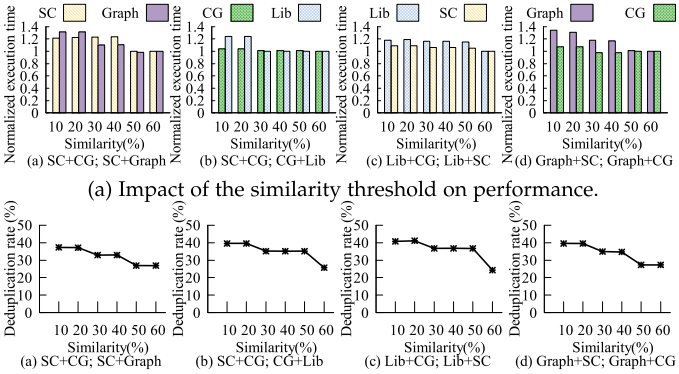


Fig. 17. Exp#7: impact of the similarity threshold.

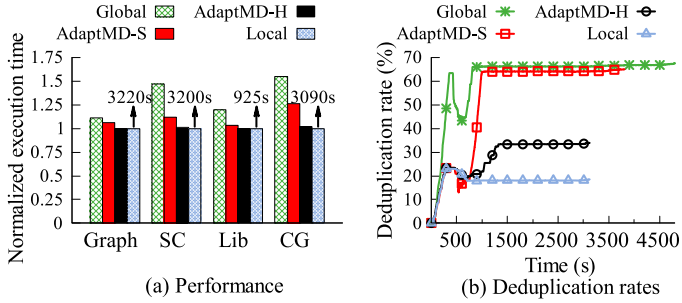


Fig. 18. Exp#8: impact of memory usage.

pages as cold, and hence reduces the deduplication rate. Thus, we set the default coldness threshold as one (Section III-B) (i.e., a page is taken as cold as long as it is not accessed in one scan period). This experiment also explains why AdaptMD-H incurs no performance overhead but has limited memory savings.

Experiment 7: Impact of the similarity threshold. We study the impact of the similarity threshold of AdaptMD-S by varying it from 10%-60% (even if two VMs run the same kernel and application, the similarity is often less than 60%). In the interest of space, we show only the experimental results under one VM deployment scenario. Specifically, we consider four applications, SC, CG, Graph, and Lib, and deploy four VMs in two NUMA nodes, i.e., one NUMA node runs two VMs, and each VM runs an application. We also consider four different configurations to verify the impact of similarity thresholds: (i) {SC, CG}, {SC, Graph}, (ii) {SC, CG}, {CG, Lib}, (iii) {Lib, CG}, {Lib, SC}, (iv) {Graph, SC}, {Graph, CG}. Note that these configurations can represent the similarity cases that run the same application or run different applications. Fig. 17 shows that as the similarity threshold increases, the normalized execution time decreases, while the deduplication rate also reduces. The reason is that a higher threshold reduces the chance of migration even if the VMs are similar. For example, when the similarity threshold is 50%, the similarity of the two VMs running SC is less than 50%, so no VM migration occurs. Through experimental results, we can see that the similarity of two VMs running the same application is usually between 30% and 50%, so we set the similarity threshold of AdaptMD-S to 40%. In addition, we find that the trade-off between memory savings and memory access performance does not change in a dramatic way, implying that the similarity threshold is not a very sensitive parameter.

Experiment 8: Impact of memory usage. To verify the effectiveness of AdaptMD-H and AdaptMD-S can be applied to large applications with more memory usage, we increase the scale of the data processed by Graph, SC and CG to increase their memory usage. After adjustment: (i) the scale and edgefactor of the graph processed by Graph are 22 and 12 respectively, and its memory usage is 3.30 GiB, (ii) the number of data points processed by SC is 2665536, and the memory usage is 4.47 GiB, (iii) the problem size processed by CG is 450000, and the memory usage is 6.02 GiB. In addition, we increase the size of memory that each virtual machine can use to 8 GiB. In the interest of space, we show only the experimental results under the mirror deployment (Section IV-A). Note that 16 VMs under the mirror deployment occupy 55.2% (i.e. 69 GiB) of the host's total memory after increasing the data scale. As shown in Fig. 18, we can see that AdaptMD-H and AdaptMD-S still effectively balance the memory savings and memory access performance in NUMA architecture with large-scale applications and large-scale memory usage.

TABLE III
EXP#9: CPU UTILIZATION (% OF CPU CYCLES)

Without bloom filter (BF)	52%
BF with page content-based hashing	57%
BF with two-level hashing	41%
Local deduplication	37%

Experiment 9: Overhead analysis. For the CPU overhead of AdaptMD-H caused by two-level hashing with Bloom filters, we consider three baselines: (i) B1, which uses red-black trees as in the KSM without Bloom filters, (ii) B2, which uses Bloom filters with three hashes to hash the page content, and (iii) B3, which does not introduce AdaptMD-H and only uses local deduplication. To study the CPU overhead in practical systems, we configure the scanning speed by using the setting in [15], [32], in which we set `pages_to_scan` as 1,000 and `sleep_millisecs` as 20 ms. Table III shows the average utilization of one CPU core. The lookups without Bloom filters (i.e., B1) consume 52% of CPU cycles of one core. If we add Bloom filters with three hashes to hash the page content (i.e., B2), the percentage of CPU cycles increases to 57%. By using Bloom filters with two level-hashing, AdaptMD-H reduces the percentage of CPU cycles to 41% as two-level hashing hashes the checksum rather than the page content.

Compared to local deduplication (i.e., B3), AdaptMD-H with two-level hashing increases CPU overhead by 4%, which is usually acceptable. For CPU overhead of AdaptMD-S, our experiment shows that AdaptMD-S costs around 80% of the cycles of one CPU core, which are mainly costed by bitmap comparison and page migrations. We emphasize that the execution time of

AdaptMD-S is not long, and it takes about 7 seconds to migrate a VM (including bitmap construction, similarity comparison, CPU/memory migration, etc.). Once AdaptMD-S is finished, AdaptMD will execute the global deduplication to eliminate page redundancy, and will trigger AdaptMD-S again only after a user-specified scan period.

For memory overhead, each page costs two bytes, one byte for Bloom filters and one byte for the inactive counter (Section III-B). Suppose that the page size is 4 KiB. The memory overhead in AdaptMD-H is 0.05%. For bitmaps of AdaptMD-S, as we only need to construct one bitmap (Section III-C) for each VM and the length is set as the maximum number of pages in all VMs, the memory overhead is just one bit for each 4 KiB page. Note that the memory occupied by bitmaps is dynamically allocated when AdaptMD-S is triggered to execute, and it is freed after execution.

V. Related Work

Memory deduplication. Existing studies mainly focus on optimizing the deduplication process, with the goals of achieving faster deduplication and higher memory savings. Specifically, hint-based solutions leverages the I/Os of VMs or the page sharing behaviors to deduplicate short-lived or file pages for more memory savings [26], [27], [39].Group-based approaches [10], [21] group VMs by users to ensure isolation and reduce page comparisons. Classification-based methods [8], [36] reduce the number of searched pages for deduplication by leveraging page types and access characteristics. Some previous studies [42], [43] optimize the data structures used for deduplication, so as to avoid unnecessary page comparisons to accelerate the deduplication. SmartMD [15] addresses the space-performance trade-off for the deduplication in huge page systems. The above studies do not consider memory deduplication in NUMA architectures. nuKSM [32] optimizes the deduplication performance in NUMA architectures, while its design is based on global deduplication only. In contrast, AdaptMD is an adaptive scheme to support different deployment scenarios to address the space-performance trade-off.

Optimizations on NUMA architectures. Many studies focus on improving the memory access performance in NUMA architectures. One research direction is to improve data locality by keeping the running processes/threads close to memory [1], [12], [33]. For example, Mitosis [1] reduce memory access to the cross-node page table through page table migration and replication. Some studies optimize the memory access performance by mitigating the contention in memory controllers or interconnect links [5], [9]. In contrast, AdaptMD focuses on memory deduplication in NUMA and works to address the space-performance trade-off.

VI. Conclusion

AdaptMD is an adaptive memory deduplication system designed to address the trade-off between memory savings and access performance in NUMA architectures. AdaptMD builds on two deduplication techniques to limit remote memory access after deduplication: AdaptMD-H leverages hotness awareness and applies global deduplication to cold pages only, and AdaptMD-S migrates similar applications to the same NUMA node for local deduplication. We further design an adaptive scheme that allow AdaptMD to be configured with the proper deduplication policies for different application deployment scenarios. Experiments on our AdaptMD prototype demonstrate the effectiveness of AdaptMD in balancing the space-performance trade-off for memory deduplication in NUMA architectures.

References

[1] R. Achermann, A. Panwar, A. Bhattacharjee, T. Roscoe, and J. Gandhi, "Mitosis: Transparently self-replicating page-tables for large-memory machines," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2020.

[2] J. Ahn, C. Kim, J. Han, Y. Choi, and J. Huh, "Dynamic virtual machine scheduling in clouds for architectural shared resources," in *Proc. USENIX Workshop HotCloud*, 2012, pp. 1–5.

[3] H. Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2020, pp. 843–857.

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

[5] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *Proc. Conf. Parallel Archit. Compilation Tech.*, 2010, pp. 557–558.

[6] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 12, no. 7, pp. 422–426, 1970.

[7] M. Boehm et al., "SystemML: Declarative machine learning on spark," *VLDB Endowment*, vol. 9, no. 13, pp. 1425–1436, 2016.

[8] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao, "CMD: Classification-based memory deduplication through page access characteristics," in *Proc. ACM SIGPLAN Notices*, 2014, pp. 65–76.

[9] M. Dashti et al., "Traffic management: A holistic approach to memory placement on NUMA systems," *SIGPLAN Notices*, vol. 48, no. 4, pp. 381–394, Mar. 2013.

[10] Y. Deng, C. Hu, T. Wo, B. Li, and L. Cui, "A memory deduplication approach based on group in virtualized environments," in *Proc. IEEE Int. Symp. Service-Oriented Syst. Eng. (SOSE)*, 2013, pp. 367–372.

[11] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *Comput. Commun. Rev.*, vol. 28, no. 4, pp. 254–265, 1998.

[12] J. Funston et al., "Placement of virtual containers on NUMA systems: A practical and comprehensive model," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, pp. 281–294, 2018.

[13] Graph500. Accessed: Nov. 2022. [Online]. Available: https://graph500.org/?page_id=12

[14] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. Shin, "Efficient memory disaggregation with INFINISWAP," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation (NSDI)*, 2017, pp. 649–667.

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

[16] J. Guo et al., "Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces," in *Proc. Int. Symp. Quality Service (IWQoS)*, 2019, pp. 1–10.

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

[18] C. Hollowell, C. Caramarcu, W. Strecker-Kellogg, A. Wong, and A. Zaytsev, "The effect of NUMA tunings on CPU performance," *J. Phys. Conf. Ser.*, vol. 664, no. 9, Dec. 2015, Art. no. 092010.

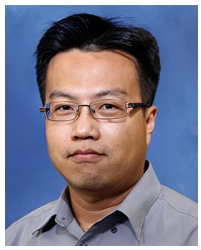
[19] "Memory latency checker." Intel. Accessed: Nov. 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[20] "Intel Resource Director Technology." Intel. Accessed: Nov. 2022. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html

[21] S. Kim, H. Kim, J. Lee, and J. Jeong, "Group-based memory oversubscription for virtualized clouds," *J. Parallel Distrib. Comput.*, vol. 70, no. 4, pp. 2241–2256, 2014.
[22] Y. Kwon, H. Yu, S. Peter, C. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *Proc. USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2016, pp. 705–721.
[23] B. Lepers, V. Quema, and A. Fedorova, "Thread and memory placement on NUMA systems: Asymmetry matters," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2015, pp. 277–289.
[24] Liblinear. Accessed: Nov. 2022. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/liblinear/
[25] P. Markthub, M. Belviranli, S. Lee, J. Vetter, and S. Matsuoka, "DRAGON: Breaking GPU memory capacity limits with direct NVM access," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2018, pp. 414–426.
[26] 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 *Proc. ASPLOS Workshop Runtime Environ., Syst., Layering Virtualized Environ. (RESoLVE'12)*, 2012.
[27] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "XLH: More effective memory deduplication scanners through cross-layer hints," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2013, pp. 279–290.
[28] A. Müller, M. Kopera, S. Marras, L. Wilcox, T. Isaac, and F. Giraldo, "Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 2, pp. 411–426, 2019.
[29] K. Narra, Z. Lin, M. Kiamari, S. Avestimehr, and M. Annavaram, "Slack squeeze coded computing for adaptive straggler mitigation," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2019, pp. 1–16.
[30] "NAS parallel benchmarks." NAS. Accessed: Nov. 2022. [Online]. Available: https://www.nas.nasa.gov/software/npb.html
[31] P. Pan, C. Li, and M. Guo, "CongraPlus: Towards efficient processing of concurrent graph queries on NUMA machines," *IEEE Trans. Parallel. Distrib. Syst. (TPDS)*, vol. 30, no. 9, pp. 1990–2002, 2019.
[32] A. Panda, A. Panwar, and A. Basu, "nuKSM: NUMA-aware memory de-duplication on multi-socket servers," in *Proc. Conf. Parallel Archit. Compilation Tech. (PACT)*, 2021, pp. 258–273.
[33] A. Panwar et al., "Fast local page-tables for virtualized NUMA servers with vMitosis," *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2021, pp. 194–210.
[34] K. Parasyris et al., "HPAC: Evaluating approximate computing techniques on HPC openMP applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2021, pp. 1–14.
[35] Z. Ruan, M. Schwarzkopf, M. Aguilera, and A. Belay, "AIFM: High-performance, application-integrated far memory," in *Proc. USENIX Symp. Oper. Syst. Des. Implement.*, 2020, pp. 315–332.
[36] S. Sha, J. Li, N. Li, W. Ju, L. Cui, and B. Li, "SmartKSM: A VMM-based memory deduplication scanner for virtual machines," in *Poster presented at SOSP*, 2013, pp. 1–2.
[37] Rodinia. [Online]. Available: http://rodinia.cs.virginia.edu/doku.php?id=start
[38] F. Trahay, M. Selva, L. Morel, and K. Marquet, "NUMAMMA: NUMA memory analyzer," in *Proc. Int. Conf. Parallel Process.*, 2018, pp. 1–10.
[39] T. Veni and S. Bhanu, "MDedup++: Exploiting temporal and spatial page-sharing behaviors for memory deduplication enhancement," *Comput. J.*, vol. 59, no. 3, pp. 353–370, 2016.
[40] C. Waldspurger, "Memory resource management in VMware ESX server," *Oper. Syst. Rev. ACM*, vol. 36, no. SI, pp. 181–194, 2002.
[41] S. Wu, H. Sun, L. Zhou, Q. Gan, and H. Jin, "vProbe: Scheduling virtual machines on NUMA systems," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, 2016, pp. 70–79.
[42] N. Xia, C. Tian, Y. Luo, H. Liu, and X. Wang, "UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 325–340.
[43] L. You, Y. Li, F. Guo, Y. Xu, J. Chen, and L. Yuan, "Leveraging array mapped tries in KSM for lightweight memory deduplication," in *Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS)*, 2019, pp. 1–8.
[44] X. You, H. Yang, Z. Luan, D. Qian, and X. Liu, "ZeroSpy: Exploring software inefficiency with redundant zeros," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2020, pp. 1–14.

Lulu Yao is currently working toward the Ph.D. degree with the School of Computer Science and Technology, University of Science and Technology of China. His research interests include virtualization and operating system, especially in memory systems.

Yongkun Li (Member, IEEE) received the B.Eng. degree in computer science from the University of Science and Technology of China (USTC), in 2008, and the Ph.D. degree in computer science and engineering from The Chinese University of Hong Kong, in 2012. He is currently an Associate Professor with the School of Computer Science and Technology, USTC. His research interests include memory and file systems, including key-value systems, distributed file systems, as well as memory and I/O optimization for virtualized systems.



Patrick P. C. Lee (Senior Member, IEEE) received the B.Eng. degree (first class honors) in information engineering from The Chinese University of Hong Kong, in 2001, the M.Phil. degree in computer science and engineering from The Chinese University of Hong Kong, in 2003, and the Ph.D. degree in computer science from Columbia University, in 2008. He is now a Professor with the Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include various applied/systems topics including storage systems, distributed systems and networks, operating systems, dependability, and security.

Xiaoyang Wang received the bachelor's degree in computer science and technology from the University of Science and Technology of China (USTC), in 2020. He is currently working toward the Ph.D. degree with the School of Computer Science and Technology, USTC. His research interests include distributed operating system, especially memory management for various systems like GPU computing, RDMA remote memory, and so on.

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