# FlexMem: Proactive Memory Deduplication for Qcow2-Based VMs with Virtual Persistent Memory

Weinan Liu
School of Informatics, Xiamen University
Xiamen, China
wnliu@stu.xmu.edu.cn

Zhihao Zhang
School of Informatics, Xiamen University
Xiamen, China
zhihaoz@stu.xmu.edu.cn

Xiangrong Liu
School of Informatics, Xiamen University
Xiamen, China
xrliu@xmu.edu.cn

Yiming Zhang
Xiamen Key Laboratory of Intelligent Storage and Computing
Xiamen, China
sdiris@gmail.com

## ABSTRACT

Virtualization-based cloud computing has gained popularity owing to its system-isolation security and near-native performance. However, a critical challenge for cloud computing arises from the presence of numerous redundant memory pages across independent virtual machines (VMs), significantly reducing memory efficiency. For instance, each VM typically loads data from its virtual disk into its memory without considering the existing data in other VMs' memory. The state-of-the-art solution for addressing memory efficiency is kernel same-page merging (KSM), which performs memory deduplication by merging pages with identical data from different processes on the host. However, KSM operates reactively after memory redundancy has occurred, adding computational overhead due to periodic scanning, rendering it inefficient for cloud computing environments with high-density VMs.

In this paper, we propose FlexMem, a preliminary architecture designed to eliminate memory redundancy for VMs. FlexMem's core is a virtual persistent memory (PMem) device with flexible backing specifications, which is designed to be memory-mapped (mmapped) to sparse and layered qcow2 images. By providing images to VMs in the form of FlexMem, and with the guest's Direct Access (DAX) enabled, user programs within the guest gain direct access to identical images already cached in the host's memory. This proactive approach eliminates memory redundancy before VMs' startup. Our evaluation demonstrates that FlexMem significantly improves memory efficiency and the availability of VMs with reduced hardware costs, ultimately enhancing the competitiveness of cloud vendors.

## 1 INTRODUCTION

Virtualization-based cloud computing, provided by leading cloud service providers such as AWS [8], Azure [14], GCP [11] and Aliyun [6], has gained widespread popularity due to its ability to provide system-isolation security and near-native performance for virtual machine (VM). Cloud data centers leverage VMs to efficiently harness hardware resources, enabling the execution of multiple workloads on a single physical machine and delivering high-availability cloud services.

A significant challenge in cloud computing lies in the presence of numerous redundant memory pages across independent VMs, leading to decreased memory efficiency in large-scale VM deployments. For instance, the page cache, an inherent operating system (OS) mechanism designed to cache file content in main memory to minimize I/O requests, tends to utilize available free memory extensively. Unused pages in the page cache are never evicted unless there is insufficient free memory for allocation. In the case of large-scale VM deployments, each VM's page cache may load identical content from the same image, without considering the existing content in other VMs' memory, resulting in significant memory redundancy across VMs.

A related work called VSwapper [7] focuses on addressing VM performance degradation under high memory pressure and attributes it to *uncooperative swappings*, such as *silent swap writes*, where unmodified pages from guest images are swapped out and two copies of these pages exist in the host disk. VSwapper tracks VM's I/O requests and directly mmaps the destination pages in the guest page cache to the identical image file on the host, which can effectively eliminate the page cache redundancy. However, there is an up-limit for the number of mappings (`DEFAULT_MAX_MAP_COUNT` for Linux [19]). As the number of I/O requests made by the VM increases, the number of mappings may also increase, which could eventually reach the limit and cause the VM to crash. Another state-of-the-art solution for memory redundancy is the kernel same-page merging (KSM) [18], a mechanism that periodically scans physical pages to find pairs of pages containing identical data. When such pairs are found, they are merged into a single page mapped into both locations to deduplicate memory redundancy among VMs. However, memory redundancy may still occur initially after starting up a VM and then gradually be deduplicated over time. This process can result in additional overhead in terms of CPU and memory usage, as well as downtime during scanning. The reactive approach can negatively impact the stable performance of physical machines, making it inefficient for cloud computing environments with high-density VMs.

The emergence of persistent memory (PMem) devices introduces a new type of memory-like storage, providing an opportunity to enhance VSwapper. Therefore, in this paper, we propose FlexMem, an architecture designed to eliminate memory redundancy for VMs. FlexMem's core is a virtual PMem device (rather than a real PMem) with flexible backing specifications, which is designed to be mmapped to sparse and layered qcow2 image. By providing images in the form of FlexMem, the page cache redundancy issue can be proactively eliminated before redundancy occurs. With guest's Direct Access (DAX) enabled, when VMs access the same offset in FlexMem, which is mapped to the corresponding offset of the image file, they access the identical physical page in the host. Any unmodified data in the image, including the read-only data and code in all the libraries and executables, can be shared among all the VMs on the host machine.

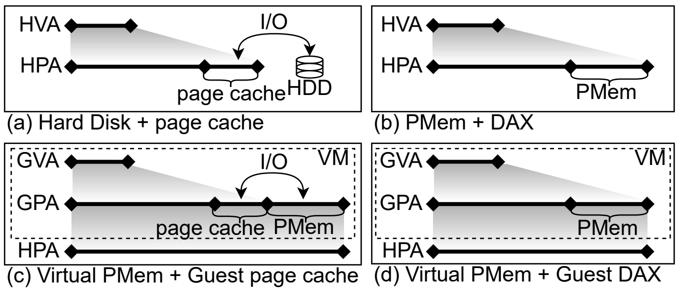
Our evaluation demonstrates that FlexMem significantly improves memory efficiency, with a notable ~35% reduction in memory usage. Furthermore, there is a considerable ~48% increase in the maximum number of VMs that can be hosted on a single machine, indicating a substantial improvement in overall VM density.

## 2 BACKGROUND

### 2.1 Virtualization in Cloud Computing

Cloud data centers typically leverage VMs to efficiently utilize hardware resources and deliver high-availability cloud services. The key to run a VM is to guarantee the effects of CPU instructions, i.e., to emulate a CPU core known as a virtual CPU (vCPU). Hardware virtualization, as an implementation of vCPU, entails direct execution of guest[1] code by the physical CPU. Modern CPUs support hardware virtualization by implementing an extend virtualization instruction set, such as Intel's `VT-x` [12], which defines two additional guest modes to execute the guest code, namely the *guest supervisor mode* and *guest user mode*, distinct from the existing *host supervisor mode* and *host user mode*. Additionally, there are several points regarding the virtualization instruction set:

[1]The term *guest* refers to the software running on VM, and the term *host* refers the machine and software supporting the VM.



(a) Hard Disk + page cache (b) PMem + DAX (c) Virtual PMem + Guest page cache (d) Virtual PMem + Guest DAX

**Figure 1: Evolutions of architectures after adding PMem and DAX and virtualization. (a) A normal situation where the storage is a hard disk and the page cache mechanism caches the data. (b) The hard disk is replaced with a PMem and the page cache is bypassed. (c) Virtualize *PMem + un-bypassed page cache* to be as the *virtual PMem + guest un-bypassed page cache.* (d) With guest's DAX enabled, and guest user programs directly access HPA space through the virtual PMem.**

- Only programs executing in host supervisor mode are permitted to enter the guest modes.
- A new type of VM page tables, such as Intel's Extended Page Tables (`EPT`) in `VT-x`, is maintained to map the guest physical address (GPA) to the host physical address (HPA) [2] for memory access within the VM.
- When interrupts and exceptions occur in guest modes, CPU traps into host supervisor mode and transfers control to the host interrupt service routine.

It is worth noting that a thread of user programs aligns with these three points, including the fact that host virtual address (HVA) is mapped to HPA, leading to *the insight that a vCPU closely resembles a thread of a user program*. Consequently, VMs can access hardware resources with almost the same capability as user programs, thus providing performance close to direct host execution for both computation-intensive and I/O-intensive workloads. Moreover, VMs offer system-level isolation, which differs from the process-level isolation provided by containers like Docker [10], leading to enhanced security. As a result, commodity cloud providers typically choose hardware virtualization to build their infrastructures and provide virtualization-based cloud services, where they create a new VM for each tenant's program and execute it within that VM.

### 2.2 Page Cache & DAX & Virtual PMem

Page cache is an inherent OS mechanism that utilizes memory to cache file contents and improve I/O performance.

[2]By the way, GVA is short for *guest virtual address* and HVA is short for *host virtual address*.

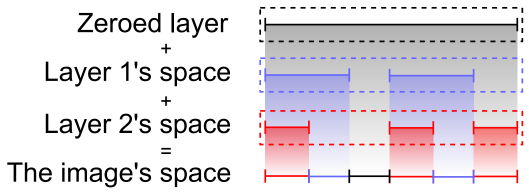


**Figure 2: An example of a two-layer sparse and layered image. For the sparse feature, each layer stores the differences in the backing layer. Regarding the layered feature, the data within the image space is composed of all its layers combined.**

Subsequently, all I/O requests to a file are first directed to the cached content in memory. By maintaining consistency between the pages in the page cache and the corresponding file content on the storage device, the page cache mechanism enables user programs to map their virtual address regions to the file's logical addresses, merging the file and memory into a unified user space and simplifying file operations, as shown in Figure 1 (a). Additionally, multiple user programs can map their virtual pages to the same file, i.e., share the same pages in the page cache, facilitating an efficient means of implementing shared memory for inter-process communication.

With DAX-enabled [17] file system, the virtual addresses can be mapped to the physical addresses whose backing is a byte-addressable device, such as persistent memory [9], which can be accessed using memory instructions by user programs, thus bypassing the page cache, as illustrated in Figure 1 (b). From the guest's perspective, a virtual PMem is presented as a block device accessed through a designated portion of the GPA space, as shown in Figure 1 (c). Adding the guest's DAX, a virtual PMem device can be used to bypass the guest OS's page cache, as shown in Figure 1 (d). In this setup, what backs the virtual PMem, i.e. a GPA space portion, is totally determined by the host.

## 2.3 Sparse and Layered Image

Image, representing a designated address space, is widely used to serve as the backing files for virtual disks or as the root filesystem of containers. The image format commonly adopts a sparse and layered structure, as shown in Figure 2. In the sparse image format, solely non-blank blocks within the space are retained. For instance, within a virtual disk image of substantial capacity, the filesystem above the image may only contain a few small files that occupy a few non-blank blocks. With the sparse image format, the image file allocates space exclusively for the actually used blocks, thereby reducing the overhead caused by numerous unused and blank blocks.

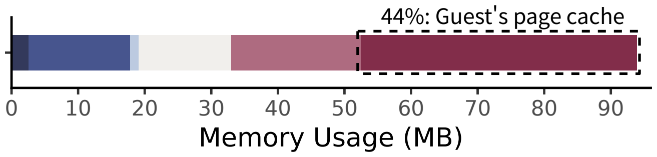


**Figure 3: The memory breakdown of a VM, utilizing an image provided with virtio-blk, reveals that 44% of the total 93 MB used memory is attributed to page cache.**

Moreover, with the incorporation of the layered feature, each layer only stores the blocks that differ from the backing layer's blocks at the same logical address, thereby eliminating redundant blocks across associated layers.

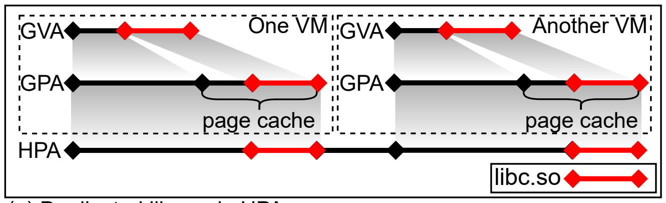
Cloud providers have recognized the efficacy of sparse and layered images [11, 13, 14], which store only the delta of the backing image. For example, tenants commonly integrate their programs into a pre-existing image containing OS distributions such as Debian [1], with their programs occupying a relatively small portion. Given that multiple tenants' programs may rely on the same distribution, only one distribution needs to be stored. This approach significantly improves efficiency in storing a vast number of images and transferring data within a disaggregated infrastructure.

## 3 MOTIVATION

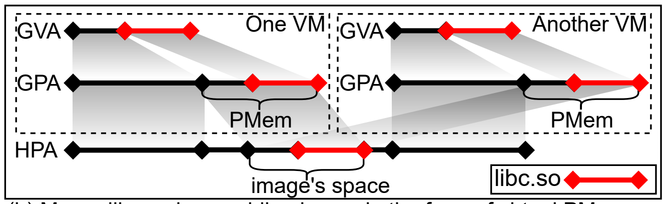
Cloud tenants often use similar applications, which has led cloud vendors to offer standardized cloud computing services. The services typically include standard images containing common server programs and runtime environments like Apache httpd [16] and NVIDIA's CUDA [15]. Tenants may also customize their images by integrating their unique contents and programs into these standard images. As a result, multiple tenants may end up using the same libraries and runtime environments from a base image, which can lead to significant redundancy in page cache usage.

Page cache of the OS is a built-in mechanism that utilizes memory to cache file contents, enhancing I/O performance. If a page in the page cache is not accessed again, the OS does not actively evict it unless there is insufficient free memory. Consequently, the page cache consumes a notable amount of memory when sufficient memory is available. For example, as depicted in Figure 3, illustrating the memory breakdown of a VM running Apache httpd without any active connections, the page cache occupies 44% of the total 93 MB used memory in the VM whose image is provided with *virtio-blk*.

In a host machine, VMs derived from the same base image may contain identical read-only data and initial states, such as the code of server programs and libraries, in their respective page caches. However, each VM's page cache operates independently, caching data without considering the



(a) Duplicated libc.so in HPA space



(b) Merge libc.so by providing image in the form of virtual PMem

**Figure 4: An example of duplicated *libc.so*. (a) Independent VMs each read libc.so into their individual page caches, leading to duplicated content on the host. (b) After enabling DAX of guest, the user programs on VMs access the content of libc.so through their respective virtual PMems, all backed by the identical image space on the host.**

existence of other VMs. Consequently, multiple duplicates of server programs and libraries coexist on the host machine, as depicted in Figure 4 (a), leading to a substantial redundancy in the page cache.

The combination of a virtual PMem (rather than a real PMem) and the guest's DAX present an opportunity to mitigate the redundancy. By using these technologies and treating vCPUs as threads, VMs can share the pages in the HPA space, as shown in Figure 4 (b). This entails loading the image into the host's memory, configuring shared pages as the backing of virtual PMems across all VMs, and enabling the guest's DAX. Consequently, all guests of users within the VMs can direct access to the executables and libraries, which exist as a single copy on the host machine. Implementing a sparse and layered image as the backing of a virtual PMem necessitates the capability of flexible memory mapping specification in the GPA region. We refer to this capability as FlexMem and will introduce it in the next section.

## 4 DESIGN

### 4.1 Image Mapping of FlexMem

FlexMem provides a flexible backing specification for delivering formatted images to VM. To specify the appropriate backing, it is necessary to know which layered image file and what offset of each block in the image's space mappes to. Algorithm 1 facilitates this determination process by systematically iterating through the blocks within the image's space.

**Algorithm 1** Algorithm for the mappings of image's space

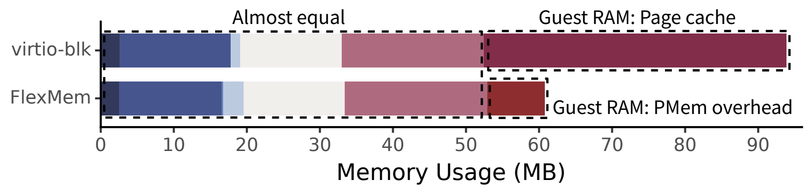
```
Input: $\langle layer_1, layer_2, ..., layer_n \rangle$ as layers of sparse images
Output: The mappings from blocks to image files
 1: $map = \emptyset$
 2: Using tmpfile to create a raw $zeroed\_layer$ whose
    length is equal to the length of $layer_1$'s space.
 3: $layers = \langle layer_1, layer_2, ..., layer_n, zeroed\_layer \rangle$
 4: for each $block$ in $layer_1$'s space do
 5:     for each $layer$ in $layers$ do
 6:         if the $layer$ stores the $block$ then
 7:             Analyzing the metadata, get the $offset$ of the
                $image$ file where the $block$ is located.
 8:             $map = map \cup \{\langle block, \langle image, offset \rangle \rangle\}$
 9:             break
10:         end if
11:     end for
12: end for
13: return $map$
```

For each block, the algorithm examines the metadata of the image to identify the layer and its corresponding offset where the block is stored. The time complexity of Algorithm 1 is $O(mn)$, where $m$ represents the length of the image's space and $n$ denotes the number of layers. This algorithm is asymptotically optimal since each block need to be mapped, and for each block, all layers need to be sequently checked to determine whether a layer stores the block.

### 4.2 Fast Dumping for Stateful VM

Stateful VMs are VMs that can preserve and subsequently dump all internal states. In order to support stateful VMs, it is necessary to identify the dirty blocks within the virtual PMem. The virtual PMem is accessed through GPA, and APIs like KVM dirty ring [21] can be used to detect the dirty blocks. The blocks can then be saved to an incremental layer based on the original layer. However, the challenge lies in preventing inconsistency during the dumping process, which could arise if a block being dumped is modified by a vCPU. A naive approach to dump VM state involves halting all vCPUs during the dumping operation, resulting in VM downtime.

Our fast method for dumping involves the following steps: (i) Pausing all vCPUs and applying write-protection to all the dirty pages in the virtual PMem. (ii) Continuing all vCPUs. (iii) Dumping the dirty pages one by one and removing the write-protection once a dirty page has been dumped. If a vCPU tries to modify a dirty page that has not been dumped yet, it will be paused. We will dump the dirty page immediately and resume the vCPU. This method minimizes VM downtime for dumping stateful VMs because vCPUs that do not touch the virtual PMem are hardly affected.



| Memory Usage (KB) | virtio-blk | FlexMem |
|---|---|---|
| ■ QEMU | 2630 | 2629 |
| ■ QEMU's stack and head | 15592 | 14316 |
| ■ The image's space | 0 | 190 |
| ■ Metadata in host kernel | 1339 | 2874 |
| ■ Guest RAM: Kernel | 14220 | 14220 |
| ■ Guest RAM: Metadata + Anonymous | 19924 | 19976 |
| ■ Guest RAM: Page cache | 42424 | 0 |
| ■ Guest RAM: PMem overhead | 0 | 8000 |

**Figure 5: VM Memory Breakdown of Two Type VMs**

## 4.3 FlexMem Daemon

It is necessary to introduce a dedicated FlexMem daemon before initializing any VM. Because without the daemon, when a layer is either compressed or encrypted, each VM would have to independently decode the data and maintain its own set of pages, thus reintroducing image redundancy within the host. The daemon can also enhance the overall system performance. This is because the Algorithm 1 needs to be executed only once for each image by the daemon, as opposed to running it separately for each image within every VM. The daemon can efficiently distribute the resulting mappings of the image's space to the VMs, with each VM only requiring $O(m)$ time to restore the mappings for FlexMem.

## 5 EVALUATION

We have implemented a prototype of FlexMem for QEMU [3], which consists of a new memory backend of the virtio-pmem [20], along with a daemon. The daemon runnes Algorithm 1 and retrieves the qcow2 image's space. The memory backend fetches the mappings to the layered images from the daemon and maps them back to a GPA region.

To demonstrate the functionality of FlexMem, we selected the official WordPress image [5] which is ~500 MB and a trimmed 14 MB Linux-6.1 kernel as the guest OS. In the following evaluations, we conduct a performance comparison between two distinct types of VMs, both configured with 96 MB of RAM. One type utilizes normal virtio-blk, while the other incorporates FlexMem. These VM instances run on a host machine equipped with dual Intel Xeon Silver 4316, each containing 20 cores, and 1 TB of RAM.

**VM instance memory breakdown.** The memory breakdown of the VM instance, as depicted in Figure 5, is obtained by analyzing `pmap` of a QEMU process and `/proc/meminfo` after 1000 VMs completed booting, showcasing the memory usage of one VM for both the guest and the host. It is worth noting that, the *Guest RAM: Page cache* in the FlexMem VM (60 MB) leads to a ~35% decrease compared to the *virtio-blk* VM (93 MB). While FlexMem can eliminate the page cache redundancy, the *Guest RAM: PMem overhead* increases due to the expansion of the GPA space for PMem. Each additional 4 KB page in the GPA space contributes to an additional 64 B `struct page` in the guest kernel and a 4 B page table entry in the host kernel, resulting in an overhead for virtio-pmem of ~1.56% for the guest and ~0.09% for the host. The memory usage of other components remains roughly unchanged.

**VM instance density.** As illustrated in Figure 6, compared to *virtio-blk* VMs, the maximum quantity of FlexMem VMs is ~48% higher on the same 1 TB RAM machine, indicating that FlexMem improves the VM density. The quantity of these two types of VM instances is roughly inversely proportional to the memory usage illustrated in Figure 5.

**VM instance boot time.** We measure the average time taken for 8 parallel VMs to complete booting simultaneously, with some VMs already having completed the booting process, as depicted in Figure 7. For both types of VMs, a pre-existing VM accelerates the VM booting by up to ~15%. The FlexMem VMs exhibit an average booting time that was ~6% faster than that of the *virtio-blk* VMs. This improvement can be attributed to the fact that the FlexMem VMs do not require loading the image content into the guests' memory. As the number of existing VMs increases, the *virtio-blk* VMs consume memory more rapidly, leading to longer boot times for subsequent VMs and eventual unavailability. Our FlexMem method improves I/O performance and significantly enhances availability at the same VM density. And when the number of VMs are between 64 and 1024, the difference in VM boot time decreased, because the overhead of FlexMem becomes more apparent (see discussion below).

**FlexMem and KSM.** We investigate memory deduplication using KSM between 2 VMs, as shown in Figure 8. In the experiment, KSM is enabled at 0 seconds. At 10 seconds, a VM is started up, followed by another VM at 20 seconds. The scanning period is set to the default value of 20 ms and the memory usage converges after ~200 s. The memory usage of *FlexMem without KSM* and *virtio-blk with KSM* both converge to approximately the same level, indicating that KSM can effectively identify the redundant guest page eliminated by FlexMem. Furthermore, the memory usage of *FlexMem with KSM* is 27 MB lower than *FlexMem without KSM*, revealing the presence of other redundant content between the 2 VMs, the majority of which is the 14 MB Linux kernel.

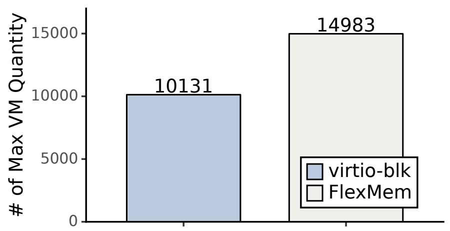


**Figure 6: VM Quantity Improvement**

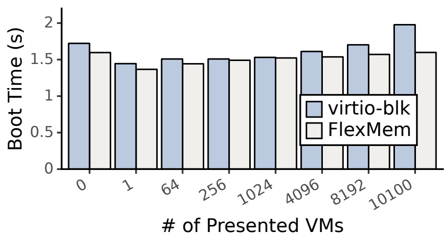


**Figure 7: VM Boot Time Speedup**

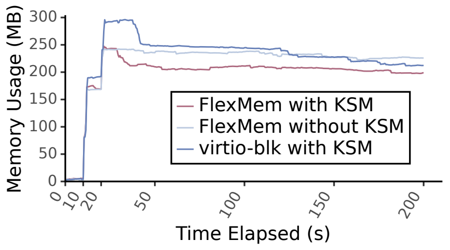


**Figure 8: Memory Usage Changes over Time with or without KSM**

## 6 DISCUSSION

**PMem overhead.** Due to the total ∼1.65% overhead of virtual PMem mentioned previously, employing a large-capacity image results in significant overhead for numerous small VM instances. To mitigate this issue, one potential solution is to eliminate the requirement for a `struct page` for the PMem device. As PMem's pages cannot be swapped out or moved, it may be unnecessary to track information about their references (i.e. reverse mapping [4]).

**Kernel sharing.** The code and data of the kernel contribute to additional memory redundancy. To address this issue, one potential solution is to interpret the kernel binary file, i.e. the ELF file [2], as a single-layer sparse image to act as the backing of FlexMem. However, this approach is akin to loading the kernel into PMem and booting from there, and requires to modify the kernel booting process.

**Memory mapping limitation.** In user space, there exists an upper limit on the number of mappings [19], defaulting to 65530. Upon reaching this limit, `mmap` returns an error. Therefore, it is essential to merge mappings from Algorithm 1 as much as possible. Additionally, the count of sparse data segments in the sparse image should not be excessively large. Fortunately, based on analysis from Alibaba's production environment [13], this count does not exceed 4500. Hence, VMs equipped with FlexMem typically do not encounter crashes due to reaching the maximum mapping count.

**FlexMem overhead.** The FlexMem overhead mainly comes from the establishment and revocation of mappings from GPA to HPA. When a VM needs to read or write a page for the first time, it requires modifying the reverse mapping, which is a mutually exclusive operation. The reverse mapping uses a red-black tree to record the virtual pages referencing the host physical page, and modifying the red-black tree incurs a logarithmic overhead. If there are thousands of VMs already referencing a host physical page, the overhead of modifying the reverse mapping is essentially constant. Additionally, the startup processes of multiple VMs can not be completely simultaneous, so FlexMem will not generate significant lock contention.

**FlexMem benefit.** As the program's heap and stack grow, the FlexMem's memory saving by deduplicating libraries and executables between VMs will gradually diminish. However, the content for a VM's startup will remain in memory, which can accelerate the startup of subsequent VMs under high memory pressure. In cloud computing scenarios where VMs frequently change, accelerating the startup effectively increases the CPU time for business programs.

**Future work.** We aim to further improve memory usage based on the aforementioned discussion. Anticipating from the VM memory breakdown, we expect that by eliminating the overhead of the PMem and implementing kernel sharing, FlexMem can achieve memory savings of up to 58% and increase the number of VMs by 140%.

## 7 CONCLUSION

In this paper, we analyze the memory redundancy in virtualization -based cloud platforms and propose a novel architecture, FlexMem, to eliminate this redundancy. With FlexMem and guest's DAX enabled, the guest redundancy can be effectively eliminated. Our evaluation demonstrates that compared to the traditional architecture, FlexMem achieves a ∼35% reduction in memory usage, and a ∼48% increase in the maximum number of VMs. The FlexMem architecture can be deployed in the cloud data center in cooperation with the remote image service to enhance VM memory usage efficiency and I/O performance, improve the availability of VMs, reduce hardware costs, and ultimately enhance the competitiveness of cloud vendors.

## ACKNOWLEDGMENTS

We thank our reviewers for their feedback. This work is supported by the National Key Research and Development Program of China (grant no. 2022YFB4500302) and the Alibaba Research Fellow Project (grant no. CCT62WHZ1220002-16611630-1). Yiming Zhang is the corresponding author.

## REFERENCES

[1] 2024. debian. (2024). https://www.debian.org/

[2] 2024. Executable and Linkable Format. (2024). https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[3] 2024. QEMU. (2024). https://www.qemu.org/
[4] 2024. rmap.c. (2024). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/mm/rmap.c
[5] 2024. wordpress. (2024). https://hub.docker.com/_/wordpress
[6] Alibaba Cloud. 2024. What is ECS? (2024). https://www.alibabacloud.com/help/en/elastic-compute-service/latest/what-is-ecs
[7] Nadav Amit, Dan Tsafrir, and Assaf Schuster. 2014. Vswapper: A memory swapper for virtualized environments. *Acm Sigplan Notices* 49, 4 (2014), 349–366.
[8] AWS. 2024. AWS Lambda. (2024). https://aws.amazon.com/lambda/features/
[9] Alexandro Baldassin, Joao Barreto, Daniel Castro, and Paolo Romano. 2021. Persistent memory: A survey of programming support and implementations. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–37.
[10] Docker Inc. 2024. docker. (2024). https://www.docker.com/
[11] Google. 2024. Google Cloud Platform. (2024). https://cloud.google.com/products/compute
[12] Intel Corporation. 2023. Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3C. (2023). https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.html
[13] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. 2020. {DADI}:{Block-Level} Image Service for Agile and Elastic Application Deployment. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 727–740.
[14] Microsoft. 2024. Microsoft Azure. (2024). https://azure.microsoft.com/en-us/products/virtual-machines/
[15] nvidia. 2024. nvidia/cuda. (2024). https://hub.docker.com/r/nvidia/cuda
[16] The Apache HTTP Server Project. 2024. httpd. (2024). https://hub.docker.com/_/httpd
[17] The kernel development community. 2024. Direct Access for files. (2024). https://docs.kernel.org/filesystems/dax.html
[18] The kernel development community. 2024. Kernel Samepage Merging. (2024). https://docs.kernel.org/admin-guide/mm/ksm.html
[19] The kernel development community. 2024. remap_file_pages() system call. (2024). https://www.kernel.org/doc/html/latest/mm/remap_file_pages.html
[20] The QEMU Project Developers. 2024. virtio pmem. (2024). https://www.qemu.org/docs/master/system/devices/virtio-pmem.html
[21] Peter Xu. 2020. KVM: Dirty ring interface. (2020). https://lwn.net/Articles/833784/