MFence: Defending Against Memory Access Interference in a Disaggregated Cloud Memory Platform

Jinhoon Lee[1], Yeonwoo Jung[1], Suyeon Lee[2], Safdar Jamil[1], Sungyong Park[1], Kwangwon Koh[3]
Hongyeon Kim[3], Kangho Kim[3], Youngjae Kim[1*]
[1]Sogang University, Seoul, Korea, [2]Georgia Institute of Technology, GA, USA, [3]ETRI, Daejeon, Korea

ABSTRACT

A VM-based disaggregated cloud memory platform (DCM) virtualizes the memory device of a remote server connected to a high-speed network as an expansion of local memory. DCM provides large memory to applications to increase throughput. However, DCM is not well-suited to managing fair memory usage between processes when they run concurrently in a VM. This is because DCM has no mechanism to provide independent memory space to each process. As a result, DCM does not guarantee fairness and performance to processes. Partitioning memory for each process is a way to solve this problem. However, in DCM, the host kernel running DCM cannot obtain the memory page information of a process (including memory page address and PID) running in the guest kernel. So it can not segregate memory pages according to the process. Therefore, this paper proposes an efficient method for the host kernel to obtain the memory page information to partition the memory for each process in DCM, called MFence. The MFence was evaluated using two Linux servers connected by a 100 Gbps IB network. Extensive evaluation has confirmed that MFence ideally provides memory partitioning to provide fairness between processes and improve overall performance.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Distributed memory**; **Process management**.

KEYWORDS

Operating System, Virtualization, Cache Partitioning

ACM Reference Format:
Jinhoon Lee[1], Yeonwoo Jung[1], Suyeon Lee[2], Safdar Jamil[1], Sungyong Park[1], Kwangwon Koh[3], Hongyeon Kim[3], Kangho Kim[3], Youngjae Kim[1]. 2023. MFence: Defending Against Memory Access Interference in a Disaggregated Cloud Memory Platform. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-March 31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3555776.3577714

*Y. Kim is the corresponding author.

1 INTRODUCTION

In-memory workloads, such as graph processing and machine learning, running in distributed clusters, are rising the popularity but are meeting a memory wall, leading to several bottlenecks, including running out of memory [1, 4, 5, 11, 12, 14, 15]. Additionally, with out-of-memory issues, clients are often required to re-execute their workloads. To overcome the out-of-memory issue, several distributed systems have deployed virtual memory using block-based storage devices. However, one of the significant drawbacks of virtual memory is expensive disk swapping, which leads to higher access latency for workloads.

Several studies have introduced remote memory access solutions, such as Fastswap [1], DCM [13], FluidMem [2], and AIFM [17], to use the memory devices of remote servers within a cluster connected through a high-speed network. The intuition is to consider the memory of a remote server as an extension of local memory to prevent memory shortage problems. Among these solutions, the VM-based distributed cloud memory platform (DCM) is a state-of-the-art VM-based remote memory solution.

A high-speed network is essential for the aforementioned remote memory access solution. However, despite a high-speed network, network data transmission delay cannot be neglected. For instance, the access latency of an application to fetch a 4KB memory page from local memory is 10 ns to 256 ns. In comparison, when the application reads the 4KB memory page from a remote server connected by EDR InfiniBand (IB) 100 Gbps, it takes 2.8 µs [8]. Because of this high remote memory access time, the remote memory access solution employs a hierarchical architecture design that aims to use local memory as an inclusive cache for remote memory and increase the hit ratio on local memory.

A user typically runs multiple processes in a VM-based server. For multiple processes, DCM's local memory is a shared cache, and processes compete for cache space [18]. If a process takes up only an unreasonably small amount of cache space, the process would unintentionally experience increased memory access times. This is the traditional shared resource contention problem. DCM (host kernel module) does not provide safeguards for managing a partitioned cache per process. For partitioned cache management, it is necessary to identify the owner/process operating on the VM (guest kernel) for each virtual memory page at the DCM level (host kernel). Therefore, this paper presents a method to identify the owner of a virtual memory page at the DCM kernel layer and explores the effectiveness of partitioned cache management for each process.

This paper makes the following contributions:

- We identified the problem of memory access interference between processes due to local shared cache contention in DCM.

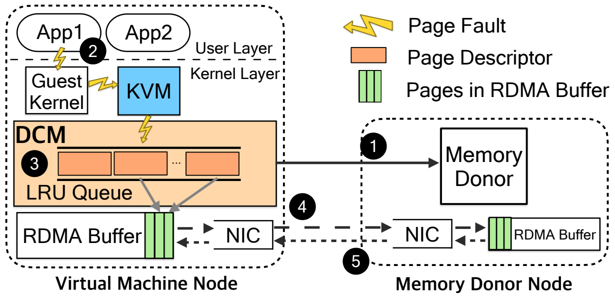


Figure 1: An overview of DCM [13].

So, we proposed MFence, which provides a fair partition of the shared cache for each process in a VM-based disaggregated cloud memory platform (DCM) [13].

- To identify the owner (process) of memory pages accessed by processes running in the guest kernel, we explored two possible memory page owner identification methods for MFence: (i) using Hypercall to inform the memory page's process ID from guest kernel to host kernel. and (ii) using guest control register 3 (gCR3), which stores the page directory address (unique per process)
- We implemented both Hypercall and gCR3 methods for MFence in DCM and experimented on two Linux servers connected by a 100 Gbps IB network. Extensive evaluations confirmed that for big data application kernels, MFence not only ensures fair cache partitioning between processes but also minimizes the unfair increase or decrease in the response time of each process, and even improve overall performance.

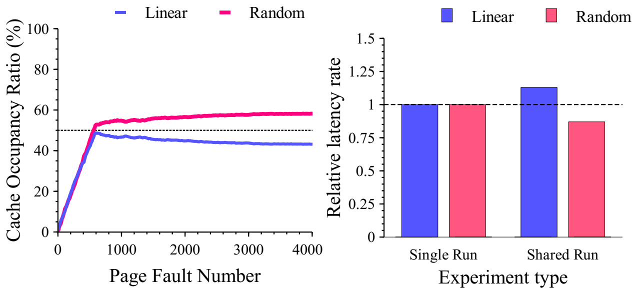
2 BACKGROUND AND MOTIVATION

This section presents the background to the disaggregated cloud memory platform (DCM) and explains the problems that arise from unfair cache sharing in DCM.

2.1 Disaggregated Cloud Memory Platform

A disaggregated cloud memory platform (DCM) [13] is a state-of-the-art VM-based remote memory solution in a virtualized environment. Figure 1 describes the hardware and software components for DCM and how they interact with each other. DCM runs on a client and server architecture. The two machines are connected by a high-speed network, such as a 100 Gbps IB network. In the figure, the client is a virtual machine node (VM node), and the server is a memory donor node. In a VM environment, the client can run multiple VMs. The VM node is divided into host and guest areas. The host area runs a hypervisor (KVM) atop the host OS through which multiple guest OSes/VMs are deployed. The guest area consists of the guest OS and applications running on it. A memory donor donates a portion of its local memory to a VM node. DCM is a Linux kernel module running in the host area of a VM node. It virtualizes the donor's local memory as if it were local memory. DCM uses local memory as an inclusive cache for remote memory to implement memory virtualization. DCM employs an LRU-based page replacement policy for cache management.

Figure 1 illustrates the operation flow of DCM, which consists of the following five steps. Step ❶ is handshaking for DCM to use the memory donor as a swap space. DCM first requests a channel connection through RDMA communication to the memory donor node to get the RDMA buffer addresses of the memory donor node. When established, a channel can perform RDMA (read/write) operations between the two. In Step ❷, when a process accesses memory, the guest OS generates a page fault if there is no page in local memory. The guest OS then forwards the page fault event to the host OS/hypervisor, which delegates page fault handling to DCM. In step ❸, DCM searches for free space in local memory first. Step ❹, if there is no free space, DCM selects pages to be replaced according to LRU policy and evicts them to the remote donor node to create new empty space. In step ❺, DCM then fetches the faulted page from remote memory to local memory via RDMA communication and adds its page descriptor to the LRU-based queue/cache.



(a) Cache occupancy (b) Relative latency

Figure 2: Unfair cache occupancy and performance gain.

2.2 Motivation

DCM improves the performance of applications that support in-memory computation by utilizing local memory as a cache. DCM adopts the LRU policy to manage the memory pages at the local memory cache. However, *a critical limitation of DCM is that processes overprovision the cache space.* As the local memory is shared between multiple processes, overprovisioning results in cache contention and an unfair cache occupancy problem.

To explain the unfair cache occupancy problem between processes in DCM, we conducted the following experiments. We created two processes where one process ran a linear pattern while the other ran random pattern workloads of Pmbench [9]. We considered two execution situations (single run and shared run). In a single run, each process runs alone on the VM where the local cache is 4GB and the memory footprint is 8GB. A shared run allows two processes to run together on the VM. Details about the experimental setup are given in Section 4.

For evaluation, we measured the local memory cache occupancy percentage for all page faults. We also defined the relative latency rate for a fair comparative evaluation. This value indicates how much response time is affected by other processes. If this value is 1, it means ideally fair memory partitioning. A value greater than or less than 1 indicates that the response time either increased or decreased due to unfair memory partitioning. Figure 2(a) shows the cache occupancy ratios at every page fault, and Figure 2(b) shows a relative latency rate of each process. From the figure, we summarize the two research motivations of this study as follows.

- **Imbalanced utilization of the shared cache:** In Figure 2(a), the local memory/cache occupancy increases until the local memory is full (i.e., around page fault number 600). After that, the cache occupancy steadily increases for the random pattern workload and decreases for the linear pattern workload. The reason is that the random workload evicts pages of the linear workload from the LRU queue since it demands caching of more memory pages than the linear workload. After that, the memory usage of the linear workload saturates to the minimum required amount to operate. As a result, the random workload occupies an average of 20% more memory than the linear workload.

- **Increased response time:** Figure 2(b) shows the relative change in latency of a shared run compared to a single run. In a single run, each process runs alone, so the relative latency rate of each is 1. On the other hand, in the shared run, since two processes are executed together, each process has a relative latency rate greater than or less than 1. The random workload improved memory access time by about 15% due to unfair cache occupation.

As observed in the analysis above, since DCM does not support cache partitioning, memory-greedy processes can occupy cache space unreasonably. However, as we explained earlier, implementing cache partitioning in DCM is a challenging task. In particular, for cache partitioning in DCM, there must be a way for the host OS to identify the owner of each memory page requested by the process running in the guest OS. Therefore, this paper presents two possible ways to identify the owner/process of each memory page and explores a cache partitioner for DCM called MFence.

3 MFENCE: DESIGN AND IMPLEMENTATION

Our proposed solution, MFence, leverages the partitioning technique to overcome the unfair utilization of the local cache between processes/applications. This section describes the design goals and details of the design and implementation of MFence.

3.1 Design Goals

In this section, we discuss our key design principles.

- **Lightweight identification of ownership of a memory page:** Each memory page's owner (process) must be identified for MFence to provide cache partitioning for each process. Since DCM operates in the host area, the page information of the process running on the guest OS/VM cannot be directly known in the host area. So, MFence designs a mechanism to identify the page of the guest process on the host. Also, identifying the owner of a page for each memory page access increases the latency of page fault handling, incurring extra overhead. Thus, MFence provides a way to identify the owner of each memory page at high speed through gCR3.
- **User-defined cache management:** DCM can provide multiple cache partitions and manage them. The user can request cache partitioning from the DCM to create and destroy a cache partition. In addition, the user should be able to command a specific process to use (or not use) a particular cache partition. For example, (i) the user requests DCM to create a cache partition with a size, (ii) the user can register a process to use the created cache, and (iii) once the process ends, the user can de-register the process to not use the cache anymore, and if none of the processes uses the cache, destroy it. So, MFence offers the API set for the user to manage the local cache by a system call.

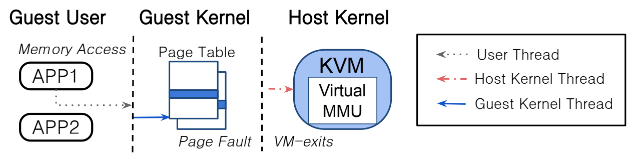


Figure 3: Description of the page fault handling in the VM.

3.2 MFence Overview

Figure 3 shows the page fault handling process in the VM. While an application is running, a page fault occurs if a mapping for the page of a specific memory address does not exist in the page table. When a page fault occurs, a context switch from the guest kernel to KVM (Hypervisor) occurs. Now, the host kernel needs to obtain the snapshot of the process which caused the page fault (e.g., the guest page address (GPA) which caused the page fault and the PID of the GPA and gCR3 register) when VM-exit occurs. However, in this situation, there is no way for the guest kernel to transmit PID along with GPA to the host kernel.

Hypercall is a method that allows the guest kernel and host kernel to communicate so that the GPA and PID can be transmitted between them. However, Hypercall is slow, but it can identify the owner who caused the page fault on a per-thread basis and its GPA. In addition, another method is using the gCR3 register of the snapshot, which can be used as a PID value. Using the gCR3 register is faster than Hypercall. Still, it cannot identify the owner on a per-thread basis but on a per-process basis. Details of each method are shown in Section 3.4.

MFence enables partitioning of the shared cache for each process. MFence consists of a cache module, the page's owner identification module, and a cache control module. The cache module manages multiple cache partitions, each implemented as a linked list-based queue. Each cache partition manages memory pages that belong to the same group of processes. Note that a cache partition can be set to manage pages of a single process or those of multiple processes in a partition group. The owner identification module identifies the page's owner fetched from the page fault handling. As explained earlier, there are two methods – Hypercall or using gCR3. The user uses the control module for cache management (resizing, allocating, and freeing cache). The user can create a region on the cache and assign a process to it. So, the partitioned cache can accommodate servicing of multiple processes.

3.3 Cache Module

MFence manages per-process LRU queue to provide different sizes of local memory partition. Initially, there is a single queue using a linked list-based data structure. When a cache partition allocation is requested, MFence partitions the queue to make a cache area for the request. MFence offers APIs for the user to allocate cache region and partition the queue to make a cache partition for the request. The partitioned queue (cache) is assigned a unique group ID (GID). Details about the GID and the APIs are shown in section 3.5.

3.4 Page Owner Identification Module

As described above, there is an LRU queue for each process. Each LRU queue caches only pages belonging to the process. To determine which queue to insert the page fetched from the remote server

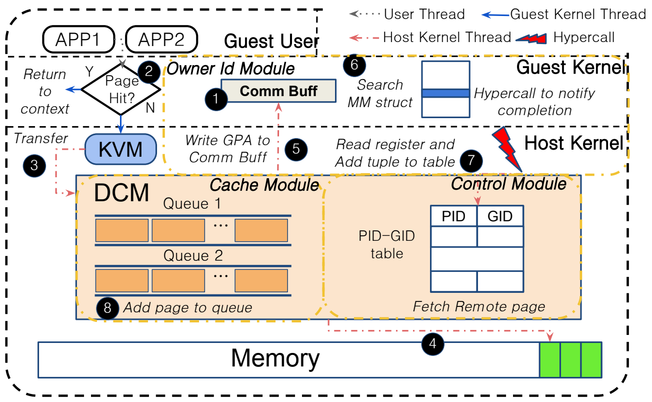


Figure 4: Description of the Hypercall method for MFence.

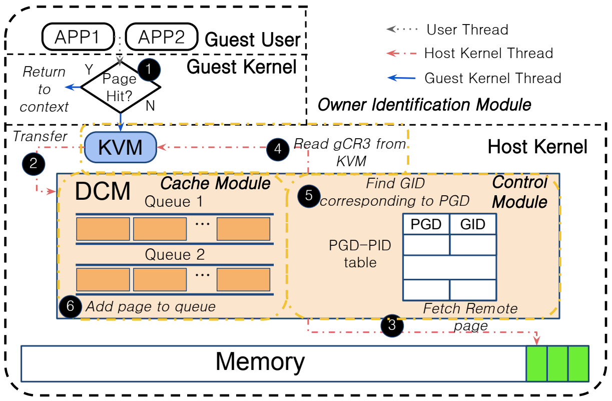


Figure 5: Description of the gCR3 method for MFence.

on a page fault, it is necessary to know the information (PID) of the process to which the page belongs at the time the page fault occurs. When a page fault occurs in the VM/guest kernel, a VM-exit and a context-switch to the host kernel occur. And DCM running on the host kernel is in charge of fetching pages from the remote server. Since the context-switch has already occurred from the guest kernel to the host kernel at the time the page fault occurred with VM-exit, the host kernel has no way of knowing information about the process that is the owner of the page that caused the page fault. We introduce two methods, using Hypercall or gCR3 value below, and explain how to determine the queue into which the fetched page should be inserted.

3.4.1 ***Hypercall method***. Figure 4 describes the Hypercall method step-by-step. First, assume that the cache is already partitioned for the process. How users/processes create their own partitioned cache in DCM and register themselves is detailed in section 3.5. The figure shows three layers: application, guest kernel, and host kernel. The KVM and DCM run on the host kernel. MFence is implemented by extending the DCM module. Specifically, the DCM module has two important data structures: a cache for GPA-PID entries and a PID-GID table. The cache minimizes Hypercall overhead, and a PID-GID table is used to maintain which process uses which cache area. A detailed description will be presented next.

❶ First, MFence create a communication buffer in the guest kernel. MFence uses an initial one-time Hypercall to provide the communication buffer address from the guest kernel to the host kernel. This communication buffer is used for information exchange between the guest and host kernels. As explained later, the host kernel writes the GPA of the page that caused the page fault in the communication buffer, and the guest kernel will read it from the buffer. ❷ When a page fault occurs in the guest kernel, a VM-exit occurs, and a context switch to the host kernel occurs. Now, the host kernel reads the register value of the vCPU to obtain the GPA address of the page that caused the page fault. The vCPU stores various information in registers, including the page's address that caused the page fault. ❸ After that, the host kernel/KVM delegates page fault handling to DCM. ❹ MFence fetches the page from the remote memory through RDMA and buffers it in the pre-reserved RDMA area. ❺ The host kernel writes the GPA to the communication buffer. ❻ Later, a context-switch from host kernel to guest kernel occurs, and the thread that has polled the communication buffer in the guest kernel begins to traverse the memory management-related kernel data structure (e.g., struct anon_vma) to obtain the PID of the corresponding GPA. Since the guest kernel has obtained the PID for the GPA, it makes a Hypercall to tell the host kernel the PID corresponding to the GPA. ❼ Lastly, MFence inserts the page fetched from the remote memory into the corresponding queue and terminates the page fault handling.

As described above, whenever a page fault occurs, the overhead involved in the Hypercall method through the communication buffer is too significant. Accordingly, MFence could minimize the overheads described above due to Hypercall by caching entries for GPA-PIDs but requires non-negligible expensive space overhead. In this work, we do not take into account the cache implementation.

3.4.2 **gCR3 method**. Figure 5 describes the gCR3 method step-by-step. Like the Hypercall method, assume that the cache is already partitioned for the process. The guest control register 3 (gCR3) method uses the address of the page global directory (PGD) of the page that caused the page fault in the host kernel as a substitute for PID. The address of the PGD can easily be obtained from the host kernel by simply reading the value of gCR3, one of several registers in the vCPU. So, unlike the Hypercall method, the host kernel does not have to communicate with the guest kernel to obtain the PID of the page that caused the page fault. Also, the gCR3 method does not manage the mapping for GPA-PID, so there is no need for a cache for them. Instead, it maintains a small table for PID-PGDs and a table for PGD-GIDs. A detailed explanation is given below.

❶ Similar to Hypercall method, when a page fault occurs in the guest kernel, the host kernel can obtain the GPA of the page that caused the page fault by simply reading the register value of the vCPU. ❷ The host kernel/KVM delegates the handling of the page fault to DCM. ❸ MFence fetches the page from the remote memory through RDMA and buffers it in the pre-reserved RDMA area. ❹ MFence reads the vCPU's gCR3 value and uses this value to determine which queue to insert the page into and then finds the PGD corresponding to the PID and consults the PGD-GID table to find the GID. ❻ Lastly, MFence fetches the buffered memory page into the corresponding queue and terminates page fault handling.

Table 1: Cache Control Module APIs for MFence.

Function	Description
Alloc_CRegion(size)	Allocate size to a new queue of the group
Add_Process_CRegion(PID, GID)	Add the process with PID to the group to GID
Remove_Process_CRegion(PID, GID)	Remove the process with PID from the group with GID
Destory_CRegion(GID)	Delete the group with to the parameter GID
Info_CRegion_All()	Show information about all cache regions, such as CRegion's GID and region size
Info_CRegion(GID)	Show information about a cache region with GID such as processes that use the cache region with GID

The associated overhead of identifying the page owner during page fault handling is small compared to the Hypercall method. But it's not free. So, like the Hypercall method, MFence can cache entries for GPA-PID, PID-GID, and GID-QID, minimizing the overhead of identifying the page owner.

3.5 Cache Control Module

The cache control module allows the user to manage the cache of that process. For example, a user can create zones in the cache and dedicate their own process to using a partitioned cache to ensure fairness and performance. Table 1 shows the list of APIs available to users. The user can use the APIs in the table to execute settings such as cache group allocation, cache area creation, and process allocation for cache groups right after a process is created or before the process completes.

User allocates cache region using Alloc_CRegion(size) and gets GID as a return value. Then, the user uses Add_Process_CRegion(PID, GID) to make the process with PID use the cache region with GID. The host kernel internally manages a table data structure (PID-GID table) of PID-GID mapping entries, so it manages which process uses which cache region. After the process is completed, the user deletes the corresponding entry from the PID-GID table using Remove_Process_CRegion(PID, GID). If there is no process mapped to the cache region with GID, the user calls Destory_CRegion(GID) to destroy the cache region. In addition, the user can check the information of cache regions using APIs such as Info_CRegion_All() or Info_CRegion(GID). MFence does not consider the handling of memory leaks caused by not calling this function.

4 EVALUATION

This section describes the experimental setup and shows how DCM outperforms traditional disk swapping on local SSDs, and the efficiency of MFence in DCM for various workload cases.

4.1 Experimental Setup

We implemented MFence in DCM [13] and evaluated it on two Intel servers running Linux, connected by a 100 Gbps IB network. The details of our experimental setup are shown in Table 2.

Table 2: Specifications of the server cluster.

CPU	Intel® Xeon Gold 6330, 2.00 GHz 28core × 2
Memory	16GB (DDR4, 3200MHz) × 8
Network	Mellanox ConnectX-5 100Gb/s EDR HCA
SSD	Intel SSD 750 (Read: 2.2 GB/s, Write: 900 MB/s) [3]
OS	Linux kernel-4.18.0-240.10.1.el8

We used representative big data application kernels [10] such as Grep, Aggregation (AG), and Group by Aggregation (GAG) and synthetic workloads using PMbench [9] for evaluation. Grep is a kernel to get the total number of occurrences of a specific word in a file of a set of words. Aggregation (AG) calculates the sum of numbers in a file of sets of numbers. Group by Aggregation (GAG) finds the sum of values of pairs with the same key in a given file with key and value pairs. We also used PMbench [9], a user-level micro-benchmark designed to measure memory access latency. PMbench generates memory-intensive workloads, referred to as linear or random workloads depending on the memory access pattern.

4.2 Evaluating the DCM

*4.2.1 **Effectiveness of DCM.***

Figure 6 shows the results of page fault frequency and average page fault latency of the application kernel by varying the local cache size. We changed the local cache size as follows:

- **L50** : Application workload size fits 50% of local cache capacity
- **L70** : Application workload size fits 70% of local cache capacity
- **L90** : Application workload size fits 90% of local cache capacity

In Figure 6, DCM shows lower page fault frequency and page fault latency overall compared to the local disk swap. The lower each value is the less page fault overhead. In all experiments, DCM's page fault overhead improvement is remarkable in all application kernels.

As shown in Figure 6(a), local disk swap in the Grep kernel shows an average page fault latency of 39 µs, while DCM shows 13 µs. In the case of page fault frequency, the local disk swap at L50 was 952,270 and 218,202 at DCM, showing a difference of about 4.36 times. In local disk swap, the page fault frequency of L70 and L90 decreased by 23% and 28%, respectively, compared to L50, but the page fault frequency of DCM is still low in L70 and L90. The Grep kernel traverses the file, reads it in character units, and stores it in a string buffer. Then, the number of words to be found is aggregated while scanning the string buffer. A page fault occurs when the string buffer size exceeds the local cache size or the memory page mapped to that word is not in the local cache while scanning the string buffer. The higher the ratio of the working set that does not fit the local cache size, the higher the page fault frequency and latency. In contrast, according to local cache size, DCM has very little change in page fault frequency and page fault latency. The different pattern of page fault overhead between the two mechanisms is due to DCM's fast paging and page prefetching effects. DCM fetches multiple memory pages from remote memory at once in case of a page fault, significantly reducing the chance of future page faults. Moreover,

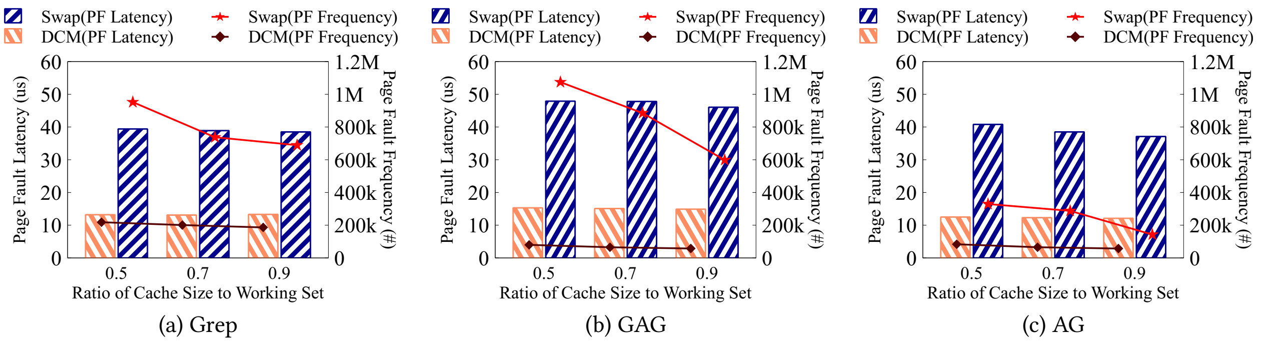


Figure 6: Page fault frequency and average page fault latency for different local cache sizes for DCM and disk swap on SSD. In the legend, PF denotes Page Fault.

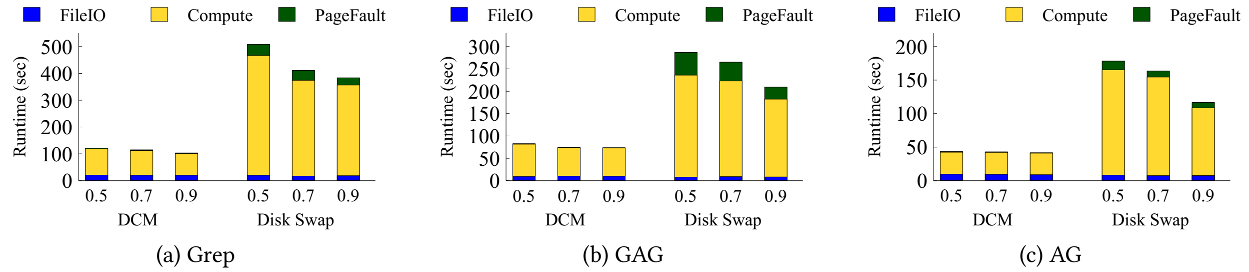


Figure 7: Time-break down analysis of a kernel runtime for DCM and disk swap on SSD. In the figure, X-axis represents the ratio of cache size to working set.

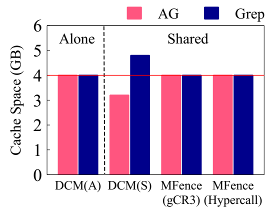
memory paging through high-speed interconnects accelerates page fault latency. These effects are prominent in the application kernel, where values corresponding to keys are randomly scattered.

Figure 6(b) and 6(c) shows the page fault overhead between two mechanisms in GAG and AG kernel. Figure 6(b), the GAG kernel shows a significant page fault overhead gap between the two mechanisms among the two application kernels. In the case of page fault frequency, the local disk swap at L50 was 1,074,375 and 79,908 at DCM, showing a difference of about 13.4 times. The GAG kernel stores many values corresponding to each duplicate key as a linked list and reads all values mapped to the key in memory during aggregation. When performing aggregation for each key, if values corresponding to the keys are randomly scattered, the page fault overhead of the disk swap increases rapidly. On the other hand, DCM fetches memory pages as large chunks during a single page fault to reduce future page faults. This reduces page fault overhead in applications that exhibit random memory patterns. This page fault overhead pattern shows a similar pattern in the AG kernel.

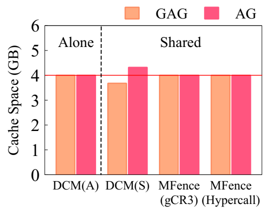
4.2.2 *Application-level Page Fault Overhead.*

Figure 7 shows the time-breakdown of each application kernel execution. We divided the overall execution time of a kernel by each operation time. We analyzed how the DCM's effectiveness affects the kernel's overall performance. All three application kernels have the following three phases: (i) *FileIO* to read files where input data is stored, (ii) *Compute* to access the data structure and perform operations, and (iii) *PageFault* to process page faults that occur during operations.

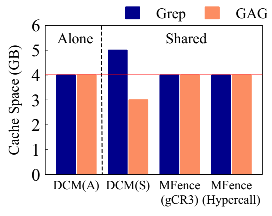
In general, a way to increase the performance of an application kernel is to quickly load data to be processed by the CPU core into memory. Figure 7 shows that the performance of an application highly depends on the ratio of *PageFault* in the application runtime. In all experiments, the ratio of *FileIO* to total runtime is consistently less than 5%. In addition, the ratio of *PageFault* to total runtime in the case of local disk swap increases as the local cache size decreases. In Figure 7(b), *PageFault* took 1.2 seconds at L50 when running the GAG kernel on the DCM, which is very short considering that the total runtime is 83 seconds. On the other hand, disk swap shows that *PageFault* accounts for about 20% of the total runtime. The GAG kernel stores the value corresponding to the key behind *FileIO* as a linked list. And when allocating memory to create a node in the list, if the local cache is full, it is allocated the remote memory space. When performing an aggregation operation, a page fault occurs if the page mapped to the value corresponding to the key does not exist in the local cache. In this case, the higher the ratio where the memory page corresponding to the values of each key does not exist in the local cache space, the higher the page fault overhead. As the ratio of *PageFault* to an application's processing time increases, application performance degradation becomes more severe in disk swap environments. Disk swap causes context switching when pages are read from the swap space through slow disk I/O, and the CPU core is blocked for a long time until the page is updated in the page table. It is analyzed that the longer the CPU core is blocked, the higher *Compute* time increases, so the application performance decreases rapidly. In contrast, DCM quickly fetches pages from



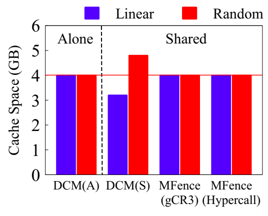
(a) AG & Grep



(b) GAG & AG

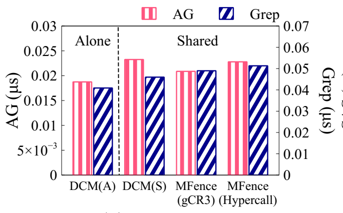


(c) Grep & GAG

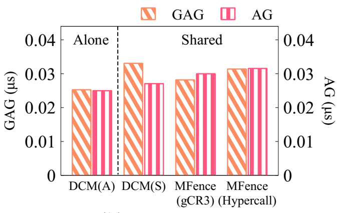


(d) Linear & Random

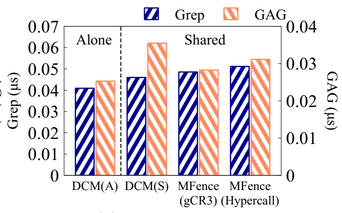
Figure 8: Memory space occupancy of each kernel for different test cases.



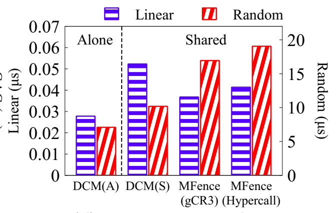
(a) AG & Grep



(b) GAG & AG



(c) Grep & GAG



(d) Linear & Random

Figure 9: Cache access latency (µs) of each process/kernel to run a kernel for different test cases.

remote memory via InfiniBand, a high-bandwidth network. Since this reduces the blocking time of the CPU core, it is seen that the CPU core performs operations quickly to increase the application performance.

Similar patterns can be observed in Figure 7(a),(c). Interestingly, the AG kernel shows a relatively low page fault overhead compared to other kernels, with *PageFault* accounting for 13%, 8%, and 7%, respectively, at L50, L70, and L90. Unlike the GAG kernel, the AG kernel scans only a single linked list that sequentially stores values corresponding to one key during aggregation. It does not require searching for duplicate keys and all mapped linked lists, so page fault overhead is relatively small. Although *PageFault* accounts for a small percentage, the slow disk I/O of local disk swap still degrades the entire application's performance.

4.3 Evaluating MFence

In order to verify the effectiveness of MFence, we compare three implementations as follows:

- **DCM(A):** DCM where the kernel is running alone. 'A' means 'Alone'
- **DCM(S):** DCM where the kernel is executed in combination with other kernels, which can cause interference between kernels. 'S' means 'Shared'
- **MFence(gCR3):** DCM with cache partitioning adopting gCR3 method
- **MFence(Hypercall):** DCM with cache partitioning adopting the Hypercall method

We evaluated MFence while run ning two or three kernels concurrently. We intended a comparison of memory access latency at the host layer (DCM). However, measuring memory access latency on every access introduces a measurement overhead. For example, putting a time-measurement function on every memory access pollutes the kernel's execution time, significantly increasing latency and deviating from accurate and fair comparisons. We devised an estimation method to compare memory access latency times at the DCM (host layer) as follows: To estimate the memory access latency, we defined Memory Access Latency as *ML*, Page Fault as *PF*, and Local Memory Hit as *MH*. We set PF rate, PF latency, and MH latency as PF_R, PF_L, and MH_L, respectively. Since the number of page faults divided by the total number of memory accesses is PF_R, ML can be calculated with the equation below.

$$ML = (1 - PF_R)\cdot MH_L + PF_R\cdot PF_L \tag{1}$$

$$PF_R = \frac{PF_{num}}{TMA_{num}}$$

We can measure the access latency and access count of local or remote memory in the DCM layer. The DCM configuration used has a block size of 32 KB that comes with a page fault. So, we measured local or remote memory access latency to handle a page fault for 32KB. In our testbed, they were 11 ns and 32 µs, respectively. And we counted the total number of memory accesses and the total number of page faults during each kernel execution.

4.3.1 *Running two kernels.*

In Figure 8 and 9, we show the results of executing two kernels simultaneously. For the experiments, the memory footprint of each workload and the total cache size are set to 8 GB. We limit a single cache partition to 4GB which is half of the cache size. Figure 8 shows the memory usage for each case. In all experiments, DCM allocates memory unfairly to either of the two kernels. On the other hand, for MFence(gCR3) and MFence(Hypercall), each kernel takes up as much as half of the local cache. For example, when the Grep and AG kernels are executed concurrently, memory space occupancy between kernels is significantly different between them (Refer to Figure 8(a)). The Grep kernel used about 1.6GB more memory than the AG workload. Because Grep demands more memory than AG, Grep generates more page faults per unit of time and takes up more cache space than AG. On the other hand, in both gCR3 and Hypercall, the local cache was equally partitioned between AG and Grep.

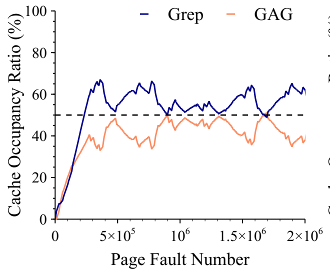
Unfairly allocated cache occupancy affects the response latency of a process. Figure 9(a) shows the memory access latency of each kernel. Comparing DCM(A) and DCM(S), AG's memory access latency for DCM(S) increased 24% compared to DCM(A). In contrast, Grep's memory access latency for DCM(S) increased by 12% compared to DCM(A). The increase in memory access latency of AG is more significant than that of Grep because, as explained earlier, Grep is allocated much more cache space than AG. On the other hand, in MFence(gCR3), it is observed that the difference in memory access latency between AG and Grep is smaller than that of DCM(S). As expected, this is due to the effect of cache partitioning by MFence. Similarly, the same observation was obtained with MFence(Hypercall). But comparing MFence(gCR3) with MFence(Hypercall), overall memory access latency is higher with MFence(Hypercall) than with MFence(gCR3). This is because the overhead of the owner identification module is higher in MFence(Hypercall) than in MFence(gCR3). As explained in Section 3.4, MFence(Hypercall) should do polling and Hypercall to communicate between host and guest. On the other hand, MFence(gCR3) has less overhead because it can simply identify the owner of the page by reading the value of the gCR3 register. Similar observations are made for GAG & AG, Grep & AG, and Linear & Random workloads.

Table 3: Analysis of performance gain using MFence(gCR3) and MFence(Hypercall) compared to DCM(S). 'L' and 'R' denote linear and random workloads in PMbench respectively.

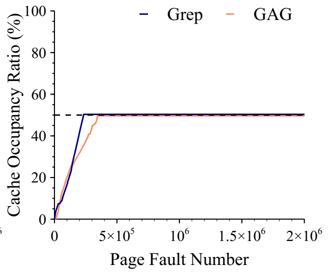
Method	AG & Grep	GAG & AG	Grep & GAG	L & R
gCR3	+5%	+7%	+19%	-24%
Hypercall	-10%	-11%	+4%	-47%

Table 3 shows the performance improvement analysis of MFence compared to DCM(S). Note that a positive value indicates an increase in performance, and a negative value indicates a decrease in performance. Since the owner identification module is not free in terms of performance overhead, we analyze the improvement of the kernel's memory access latency by MFence. Therefore, we calculated the average access latency of co-executing kernels for DCM(S) and MFence and compared them. In Table 3, MFence(gCR3) improves memory access latency by -24% to 19% compared to DCM(S). MFence(gCR3) also shows negative performance gain (Linear & Random workload). This is because PMbench is a memory-intensive workload. In this memory-intensive workload mix, cache partitioning is unnecessary because MFence is ineffective. On the other hand, as expected, the improvement in memory access latency of MFence(Hypercall) is less than that of MFence(gCR3). The improvement in memory access latency is only -47% to +4%. Therefore, MFence(gCR3) has higher performance efficiency than MFence(Hypercall).

4.3.2 ***Microscopic analysis of cache occupancy.*** We performed a microscopic analysis of cache sharing between processes. Figure 10(a) shows the change in cache share of each process on every page fault when Grep and GAG kernels are running. In DCM(S), Grep and GAG kernels compete for a shared cache. At first, they

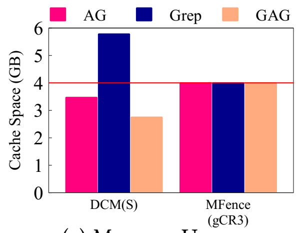


(a) DCM(S)

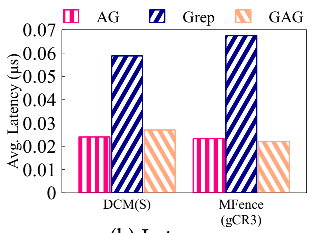


(b) MFence(gCR3)

Figure 10: Changes in cache occupancy for each process.



(a) Memory Usage



(b) Latency

Figure 11: Results when running three kernels.

occupy the cache space evenly, but after that, the Grep kernel takes up more cache space than the GAG kernel. Overall, the Grep kernel has a higher cache share than the GAG kernel, but sometimes the GAG kernel tries to steal cache space from the Grep kernel. Eventually, the cache space contention between them reveals the dynamic fluctuation of cache share over time. On the other hand, Figure 10(b) shows the perfect distribution of cache space between the Grep and GAG kernels in MFence (gCR3). Since the Grep kernel has a higher memory demand than the GAG kernel, it reaches 50% of the cache space before the GAG kernel as expected. There is no dynamic change in cache share, as shown in Figure 10(a).

4.3.3 ***Running three kernels.*** In Figure 11, we show the results of executing three kernels in parallel. For these experiments, the memory footprint of each workload and the total cache size are set at 8 GB, while we limit a single cache partition to be 1/3 of the cache size. In Figure 11, as expected, DCM(S) has an unfair cache partition between kernels. In DCM(S), Grep occupies more memory cache than AG and GAG. Figure 11(a) shows that AG uses 3.47GB, Grep uses 5.78GB, and GAG uses 2.75 GB. However, MFence(gCR3) strictly limits the cache partition so that each kernel fairly takes up its cache space. Figure 11(b) shows the memory access latency after cache partitioning by MFence(gCR3). As expected, Grep increased access latency by returning the improperly acquired cache space, while AG and GAG were allocated more cache space and slightly reduced access latency. Finally, we analyzed the overall performance improvement by MFence(gCR3), just like when we ran the two kernels earlier. The overall performance improvement of MFence(gCR3) is about 6% compared to DCM(S).

5 RELATED WORK

Resource Disaggregation: Recent studies have proposed various designs to access remote memory by proposing new network systems [7], adding user-level functions [17], modifying kernel features [1, 6], and supporting hypervisor-integrated systems for virtualized environments [2, 13]. Fastswap [1] attempted to restrict the capacity of a cache (i.e., local memory) to an application using cgroup. It allocates different sizes of cache per application, following its scheduling policy to improve the makespan of multiple applications. However, using cgroup is not applicable when applications are running on the VM, as the host kernel is not aware of individual processes on the VM of the guest kernel. It is possible to control cache size per VM, not per application on VMs. Similarly, FluidMem [2] also adjusted the cache quota per-VM basis. Still, it did not control the cache size per process. MFence proposes a mechanism of cache partitioning to control the amount of cache per application on a VM.

Unfairness of Shared Cache: FairRide [16] investigates the problem of fair allocation of cache for multiple users with shared files. It achieves an isolation cache in which users get better performance through blocking. This paper focuses on how users will ensure fairness arising from shared files. However, MFence proposes a mechanism to ensure fairness when processes run concurrently in a VM environment. In addition, BWLOCK [19] pointed out the problem of memory bandwidth contention between real-time and non-real-time applications. BWLOCK observed that due to the high memory bandwidth required by the CPU core where non-real-time applications are running, the memory bandwidth allocated to real-time applications is limited, resulting in application deadlines not being met. To ensure application deadlines, BWLOCK provides users with predefined APIs in memory-intensive code sections of real-time applications to trigger the memory bandwidth of cores running non-real-time applications. MFence also ensures memory resources without memory bandwidth throttling at runtime, as users can partition local memory per application using APIs such as command tools.

6 CONCLUSION

This paper proposes MFence to solve the problem of unfair memory occupation of shared cache between processes in a VM-based distributed cloud memory platform (DCM). MFence provides a process with an independent partitioned cache area, completely eliminating cache interference between processes. MFence uses the page global directory (PGD) address of the process that caused the page fault in the guest kernel, stored in the gCR3 register, to identify the process and the partitioned cache area to use. The extensive evaluation shows that MFence enables fair cache partitioning with low overhead and results in performance gains. As a representative example, MFence shows a 14.4% performance improvement for workloads where the AG and Grep kernels run concurrently while ensuring fair cache partitioning.

ACKNOWLEDGEMENT

This work was partly supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00503, Researches on next generation memory-centric computing system architecture), Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korea government (No.22ZS1300, Research on High Performance Computing Technology to overcome limitations of AI processing), and National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2014386).

REFERENCES

[1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. 2020. Can Far Memory Improve Job Throughput?. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys '20)*. 1–16.

[2] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. 2020. FluidMem: Full, Flexible, and Fast Memory Disaggregation for the Cloud. In *Proceedings of the IEEE 40th International Conference on Distributed Computing Systems (ICDCS '20)*. 665–677.

[3] Intel corporation. 2015. Intel® SSD 750 Series Product Specification. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-750-spec.pdf

[4] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference (EuroSys '18)*. 1–13.

[5] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *Proceedings of the USENIX Annual Technical Conference (ATC '22)*. 287–294.

[6] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)*. 649–667.

[7] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A Hardware-Software Co-Designed Disaggregated Memory System. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 417–433.

[8] Munira Hussain. 2018. *Need for Speed : Comparing FDR and EDR InfiniBand*. https://downloads.dell.com/manuals/all-products/esuprt_software/esuprt_it_ops_datcentr_mgmt/high-computing-solution-resources_white-papers77_en-us.pdf

[9] Yang Jisoo and Seymour Julian. 2018. Pmbench: A Micro-Benchmark for Profiling Paging Performance on a System with Low-Latency SSDs. In *Proceedings of the Information Technology New Generations (ITNG '18)*. 627–633.

[10] Awais Khan, Attique Muhammad, Youngjae Kim, Sungyong Park, and Byungchul Tak. 2018. EDGESTORE: A Single Namespace and Resource-Aware Federation File System for Edge Servers. In *Proceedings of the IEEE International Conference on Edge Computing (EDGE '18)*. 101–108.

[11] Awais Khan, Hyogi Sim, Sudharshan S Vazhkudai, and Youngjae Kim. 2021. Mosiqs: Persistent Memory Object Storage with Metadata Indexing and Querying for Scientific computing. *IEEE Access* 9 (2021), 85217–85231.

[12] Awais Khan, Hyogi Sim, Sudharshan S Vazhkudai, Jinsuk Ma, Myeong-Hoon Oh, and Youngjae Kim. 2020. Persistent Memory Object Storage and Indexing for Scientific Computing. In *Proceedings of the IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC '20)*. IEEE, 1–9.

[13] Kwangwon Koh, Kangho Kim, Seunghyub Jeon, and Jaehyuk Huh. 2019. Disaggregated Cloud Memory with Elastic Block Management. *IEEE Trans. Comput.* 68, 1 (2019), 39–52.

[14] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 317–330.

[15] Ling Liu, Wenqi Cao, Semih Sahin, Qi Zhang, Juhyun Bae, and Yanzhao Wu. 2019. Memory Disaggregation: Research Problems and Opportunities. In *Proceedings of the IEEE 39th International Conference on Distributed Computing Systems (ICDCS '21)*. 1664–1673.

[16] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. 2016. FairRide: Near-Optimal, Fair Cache Sharing. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*. 393–406.

[17] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. 315–332.

[18] Jun Xiao, Yixian Shen, and Andy D. Pimentel. 2022. Cache Interference-Aware Task Partitioning for Non-Preemptive Real-Time Multi-Core Systems. *ACM Transactions on Embedded Computing Systems,* 21, 3 (2022), 1–28.

[19] Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. 2017. BWLOCK: A Dynamic Memory Access Control Framework for Soft Real-Time Applications on Multicore Platforms. *IEEE Trans. Comput.* 66, 7 (2017), 1247–1252.